Rendermanía

nº 0

SUPLEMENTO MENSUAL DEDICADO A LA IMAGEN SINTÉTICA

Comandos de línea, objetos, operaciones CSG, materiales, luces, cámaras...

Todo lo que necesitas saber sobre POV 3.0

Un proyecto de altura

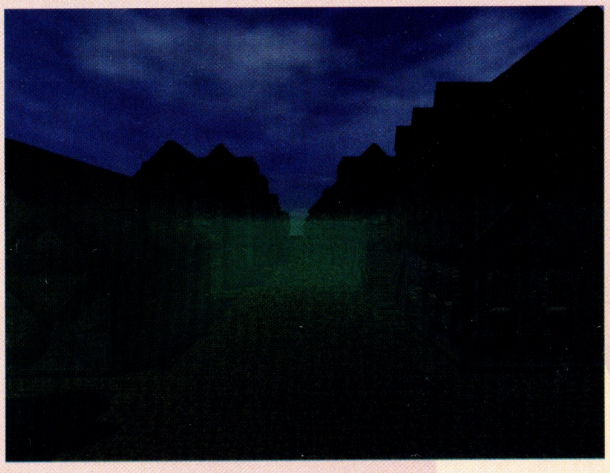
Aprende a construir edificios virtuales

El universo 3D de POV

CONTENIDO EN PCMANIA CD

El universo

Las presentes páginas deben su existencia, sobre todo, al elevado número de cartas recibidas donde los lectores reclamaban la inclusión de tal o cual apartado dentro de Rendermanía. Evidentemente, era imposible contentar a todos dentro del limitado espacio disponible en las 5 páginas de la sección y por ello PCmanía ha tomado la decisión de ampliar Rendermanía, la cual, desde ahora, pasa a ser un suplemento. Como nos gustan las sorpresas no desvelaremos los apartados de que constará la nueva Rendermanía, aunque, eso sí, confiamos que sean del gusto de todos los afectados por el virus infográfico.



Las imágenes sintéticas son generadas por programas que describen objetos 3D.

Ante todo, queremos agradecer al POV-Team la creación (y contínua mejora desde hace años) de lo que muchos pensamos que es el mejor trazador de rayos que pueda encontrarse para PC: El Persistence of Vision Raytracer (o sea POV).
También, queremos agradecer a Roberto Potenciano, José González y Héctor Iniesta que nos prestaron su ayuda con los bitmaps y dieron sus opiniones en las escenas.
Finalmente, queremos agradecer los estímulos, consejos, notas, utilidades, escenas y animaciones de todos aquellos lectores que, a lo largo del tiempo, con sus cartas y e-mails, nos han hecho comprender que la Infografía es algo más que la creación de imagen sintética. Es tambien algo que une a la gente.

de POV 3.0

Por José Manuel Muñoz

Este primer número es especial; ya que se dedica única y exclusivamente a POV 3.0. Su contenido está al alcance de cualquier novato, y se comentan características de la nueva versión 3.0, además de un pequeño proyecto realizado con dicha versión, para beneficio de los povmaniacos más veteranos.

Estas páginas han pasado por una larga evolución. En principio se pensó hacer un pequeño libro sobre POV, luego se decidió esperar al lanzamiento de la versión 3.0 y, finalmente, se tomó la decisión de crear un suplemento de infografía "práctica" cuyo primer número dedicamos al que ha sido caballo de batalla de Rendermanía durante tanto tiempo.

En cuanto a este primer número, casi todas las imágenes se han creado usando una pequeña librería que se adjunta en el CD: castlib 1.8. Se trata de una librería de formas para construir castillos, pueblos y ciudades medievales con POV 3.0. Comenzamos a desarrollar esta idea hace ya mucho -cuando los 386 dominaban la tierra- pero tuvo que abandonarse por falta de potencia del procesador, falta de memoria, falta de tiempo, y -sobre todo- por falta de ciertas sentencias en el lenguaje escénico de POV. Sólo ahora, con todas estas carencias, parcial o totalmente enmendadas, se ha podido dar a castlib un razonable grado de terminación (al menos en el apartado del pueblo). De hecho, toda la librería se reescribió varias veces bajo POV 2.2 para luego, tirarla al cubo y volverla a escribir bajo POV 3.0). Sin embargo, castlib está incompleta aún. En el momento de escribir esto, muchas piezas necesarias para crear castillos aún estaban en fase de prueba. Por ello, se ha decidido extraer un fichero con el apartado que resultaba más completo: el de las casas precisas para construir una pequeña ciudad medieval.

¿Y qué ordenador se precisa? Algunas escenas pueden requerir un 486 con 16, o mejor 32 Megas de RAM. Esto no significa, no obstante, que un usuario provisto de un ordenador menos potente haya de renunciar a seguir los ejemplos. Castlib puede ser recortada para crear escenas más sencillas.

Se entiende como "render" al proceso de generación de la imagen.

Infografía y

Podemos definir el término Infografía como la rama relativamente nueva de la informática dedicada a la generación de imágenes por ordenador.

Estas imágenes "sintéticas" son generadas por programas que utilizan datos que describen objetos tridimensionales para universos virtuales que sólo existen en los ordenadores. Los resultados son moneda corriente para todos nosotros desde hace ya años: anuncios, animaciones y clips de gran realismo aparecen de manera cotidiana en la televisión y el cine. Un ejemplo ya clásico, es la película Terminator II, donde a la imagen real se le añadía la generada por ordenador para el androide T1000. Ya antes se habían empleado efectos infográficos, pero quizá fue Terminator la película que más hizo comprender a todo el mundo las posibilidades de la infografía. Posibilidades que parecen ampliarse con cada nueva película: The Mask, Jumanji, Toy Story, Twister, Independence Day y muchas más.

De todo esto se desprende que una de las mayores ventajas de la infografía es la de permitirnos crear, con un realismo creciente, objetos que no existen en el mundo real, que están en fase de diseño, o que nunca tendrán una existencia física. Podríamos, por ejemplo, (si tenemos los programas y equipos adecuados) simular un tornado, pasear por la Constantinopla de tiempos del emperador Heráclio, recrear la vida de una hormiga, el comportamiento de un nuevo coche, el vuelo de una nave espacial, simular la evolución de una estrella, etc. Aparte de nuestra imaginación, el único límite está en la potencia de programas y ordenadores y esta potencia aumenta de día en día, lo que esta permitiendo que la infografía tenga una presencia cada vez mayor en un campo tan popular como el de los videojuegos de PC («Doom», «Duke Nukem», «Quake», etc.). Por otro lado, otra de las razones del creciente interés de la infografía es que cualquiera tiene la posibilidad de llegar a hacer cosas interesantes. En principio basta con disponer del ordenador más difundido del planeta: el PC.

La idea en que se basa en Ray-tracing consiste en "lanzar" rayos desde la cámara para cada pixel de la escena a calcular.

Diversos tipos de render

Existen muchos programas generadores de imágenes fotorealistas para todo tipo de platafomas y también

trazado de rayos

una gran cantidad de métodos o algoritmos distintos para generar imágenes por ordenador. Además, la implementación de un mismo método puede variar de un programa a otro, ya que los programadores pueden combinar diversos algoritmos de sombreado, ocultación de superficies, tratamiento de color, etc., para construir el motor 3D del programa.

(Llamamos motor 3D a la parte de código del programa encargada del proceso de Render. En cuanto al proceso de Render en sí, se entiende como tal al proceso de generación de la imagen. En POV, Polyray, 3D Studio, Imagine o cualquier otro programa de este tipo, decimos que se está efectuando el Render cuando el programa está calculando la escena, aunque ésta no sea visible en pantalla).

El trazado de rayos o Ray-tracing es, simplemente, uno de los muchos métodos existentes para la generación de imágenes fotorealistas. Programas como POV, Polyray e Imagine utilizan el trazado de rayos mientras que otros como 3D Studio no lo emplean. El trazado de rayos fue ideado, en principio, como un algoritmo de ocultación de superficies y sólo posteriormente se convirtió en un método de sombreado.

(Un algoritmo es un método para resolver un problema dado. La ocultación de superficies es uno de los problemas básicos a los que debe enfrentarse cualquier programador gráfico. Se refiere al proceso por el que se decide qué superficies quedan más cerca del observador, a fin de representarlas en el orden correcto.

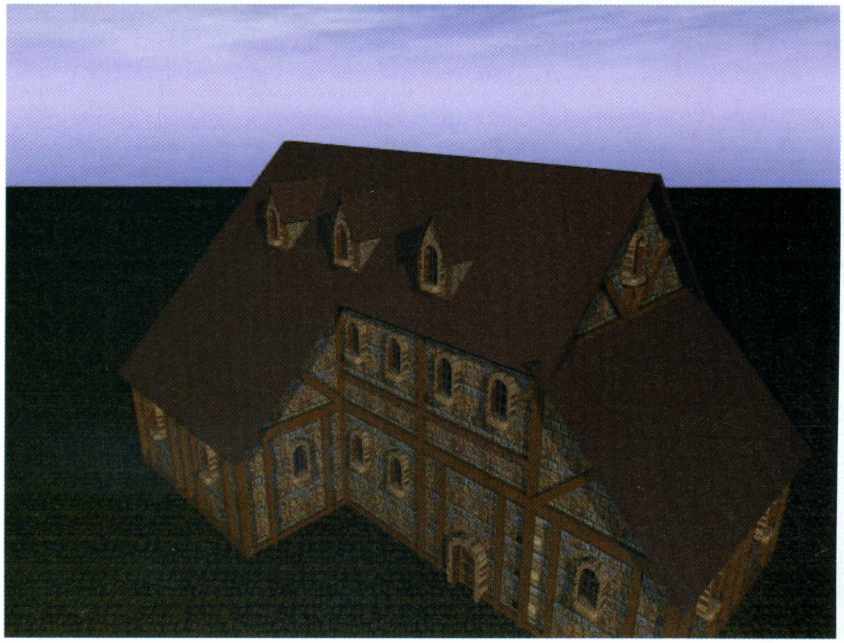
Un método de sombreado es cualquier algoritmo que emplee uno de los muchos modelos matemáticos de iluminación existentes para sombrear -dibujar-, de manera más o menos realista, las superficies).

Ray-Tracing

La idea en que se basa el Ray-tracing consiste en "lanzar" rayos desde la cámara u observador para cada pixel de la escena a calcular. Estos rayos del "observador", al viajar por el universo virtual, pueden chocar o no con un objeto. Si este choque no se produce, el pixel toma el color del fondo.

En caso contrario, el color del pixel se determinará dependiendo del color propio de la superficie, el color de las fuentes que incidan sobre él, etc.

Debido a esto, este método es llamado también "de seguimiento de rayos". En principio podría suponerse que lo lógico sería



Cuanto mayor sea la complejidad de la escena mayor será el tiempo que precisará el PC para completarla.

que el programa efectuase el seguimiento de los rayos desde las fuentes de luz de que parten estos, hasta llegar al observador. Pero esto no es práctico: habría que seguir muchísimos rayos con levísimas variaciones en el ángulo de lanzamiento para garantizar una iluminación adecuada para las superficies. Además, un porcentaje muy elevado de los rayos lanzados no resultarían útiles, esto es, no llegarían al espectador. Por ello estos rayos, a los que llamaremos principales, se lanzan desde el ojo, y se los traza siguiendo un camino inverso al que siguen en el universo real. Cuando un rayo principal choca contra la superficie de un objeto hay que decidir el color en ese punto.

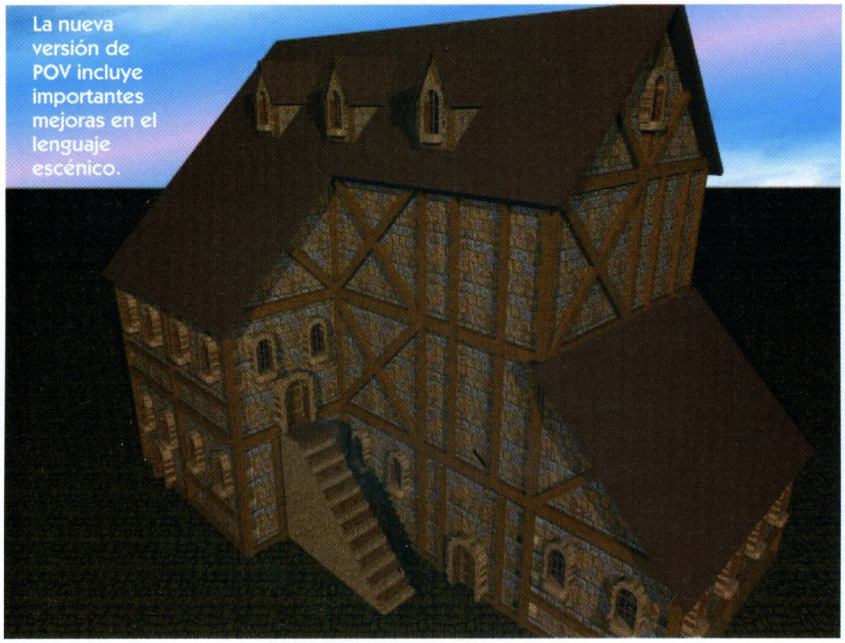
Para ello, se lanzan desde ahí otros rayos -uno para cada fuente de luz definida en la escena- para decidir si dichas fuentes inciden sobre el punto visible. También, si la superficie alcanzada es reflectante, se lanzará un rayo en la dirección de reflexión para determinar el color a reflejar.

Todo este proceso es recursivo e implica muchos cálculos. Pueden implementarse muchas variaciones al proceso descrito pero, por defecto, el trazado de rayos tiene en cuenta la suma de la luz de las diversas fuentes, el color propio de los objetos, las reflexiones, sombras, etc. Debido a todo esto, las imágenes generadas con técnicas de Ray-tracing tienen un nivel muy elevado de realismo. Otros métodos de sombreado consiguen generar imágenes en menos tiempo pero a cambio de un menor "foto-realismo".

Frecuentemente, además, el uso de estos métodos obliga al usuario a un trabajo adicional para igualar el realismo del trazado de rayos. Usando un programa de Ray-tracing bastará con establecer la colocación de las fuentes y de los objetos, el color y propiedades de las superficies, colocar la cámara, y obtendremos un resultado muy similar al que podría obtenerse con una foto de una escena similar en el mundo real.

Naturalmente no todo es de color de rosa. El modelo de iluminación del trazado de rayos funciona muy bien en casi todos los aspectos salvo en el de la luz difusa. ¿Y qué es la luz difusa? Bien: todos los objetos físicos del mundo real reflejan la luz que reciben (salvo los cuerpos celestes llamados agujeros negros y los dispositivos de camuflaje de las aves de presa Klingon, en Star Trek). Como la mayoría de las superficies del mundo real son irregulares en mayor o menor medida, los rayos reflejados no abandonan estas superficies con el mismo ángulo de incidencia con que llegaron a ella. Es esta luz difusa la que ilumina a los objetos sobre los que no caen directamente los rayos de un foco de luz. También, es esta luz la que suaviza los bordes de las sombras. (Consideremos, por ejemplo, el aspecto de un cuarto oscuro en el que se filtran los rayos de luz a traves de una persiana. Sin luz difusa no podríamos percibir a los muebles en penumbra). Para simular la luz difusa, los trazadores de rayos emplean un truco llamado iluminación ambiental, el cual, desgraciadamente, no se acerca demasiado a la realidad.

Otra limitación importante del modelo luminoso de los trazadores de rayos es que las fuentes de luz son puntos infinitamente pequeños. Esto causa problemas con las sombras ya que, al no calcularse la luz difusa, las sombras tienen unos bordes excesivamente nítidos. (Esto puede enmendarse, por ejemplo, empleando áreas de luz). Otros modelos luminosos que solucionan estas limitaciones se basan en la "radiosidad". Desgraciadamente la radiosidad, a su vez, tiene problemas con



La nueva versión de POV incluye importantes mejoras en el lenguaje escénico.

las reflexiones especulares y requiere un tiempo de cálculo considerablemente superior incluso al precisado por el Ray-tracing. Si lo mencionamos es porque POV, a partir de la versión 3.0, incorpora la radiosidad como opción.

Así, en resumen, la cantidad total de cálculos para crear una imagen es muy grande y cuanto mayor sea la complejidad de la escena mayor será el tiempo que precisará el ordenador para completarla. El proceso puede durar minutos, horas, días, o semanas enteras dependiendo también de la potencia del ordenador, o de la del propio Ray-tracing. Por esta razón, lo que acabamos de explicar sobre el trazado de rayos nos será útil más tarde, cuando estudiemos los diversos métodos existentes para ahorrar tiempo de cálculo.

¿Qué tiene POV de especial?

Persistence of Vision Raytracer, abreviadamente llamado Povray o POV, es el nombre del trazador de rayos sobre el que trata este suplemento.

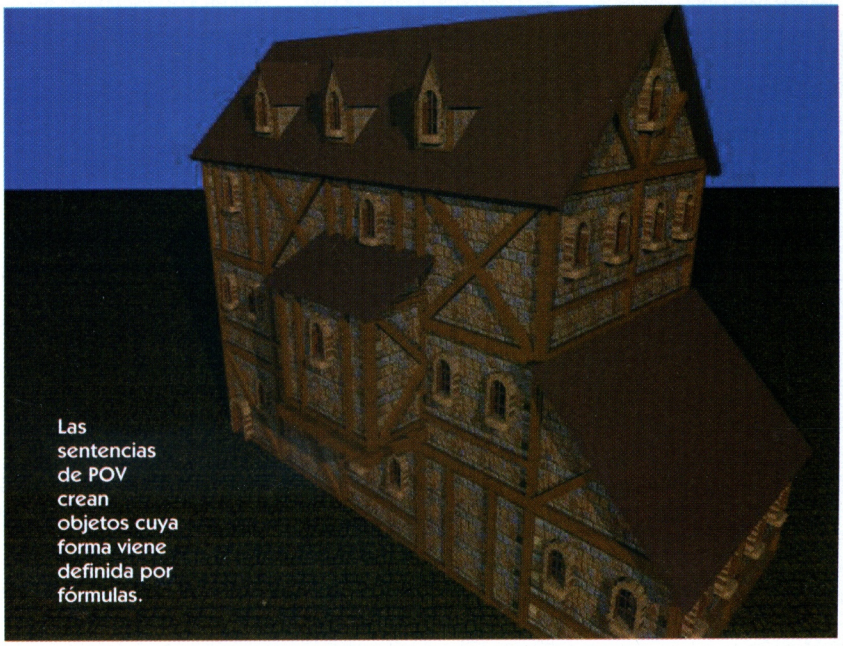
Persistence of Vision Team (o pov-team) es también el nombre del grupo de programadores que nació con la finalidad de crear el Raytracer POV. Hace ya unos cuantos años existían otros Ray-tracers para PC, recorvertidos desde amiga en su mayor parte. Pero, insatisfecho con estos programas, el grupo formado por Drew Wells decidió crear un nuevo Ray-tracer basado en el trazador de rayos DKBTrace de David Buck y Aaron Collins (los cuales se sumaron al proyecto). Con este fin se estableció el cuartel general del grupo en el foro GRAPHDEV de CompuServe y, desde entonces, se han publicado varias versiones de POV con un éxito creciente.

Actualmente, hay versiones de POV que funcionan bajo MS-DOS, Windows 3.x, 95 y NT, Linux, Apple Macintosh 68k y Power PC, Amiga, UNIX y otras plataformas. La última versión de POV la podéis encontrar en su dirección oficial vía Web en http://www.povray.org, o vía ftp, en ftp.povray.org. Pero tanto el programa como muchas de las utilidades que se han creado para él se pueden encontrar en muchas direcciones más, por todo el mundo.

¿Cuál es la razón de este éxito? En primer lugar, por supuesto, la increible calidad del programa. Los miembros del pov-team llevan años trabajando ininterrumpidamente para que POV no se quede estancado, como ha sucedido con otros trazadores de rayos que han acabado desapareciendo. En parte para garantizar esta calidad, el pov-team es un grupo abierto, donde, según parece, pueden entrar nuevos miembros, si estos tienen algo interesante que aportar al proyecto (esto fue lo que sucedió con Dieter Bayer y su FTpov). Este factor queda reforzado por el hecho de que los fuentes de POV son de libre distribución. Cualquier programador puede compilar su propia versión de POV, e incluso incorporar sus propios cambios, si tiene los conocimientos y la capacidad necesarias (aunque debe indicar que se trata de una versión no oficial).

Las características del programa que, prácticamente desde el principio, ofrecía una excelente relación tiempo/calidad, atrajeron a muchos artistas que comenzaron a publicar escenas creadas con POV. Esto a su vez estimuló la creación de utilidades específicas y la adaptación de otras para trabajar con POV.

Por último, hay que citar un pequeño detalle: POV es de libre uso. No hay que pagar un céntimo por su empleo y esto, junto a su potencia y los factores antes citados, ha creado la bola de nieve que ha hecho



Las sentencias de POV crean objetos cuya forma viene definida por fórmulas.

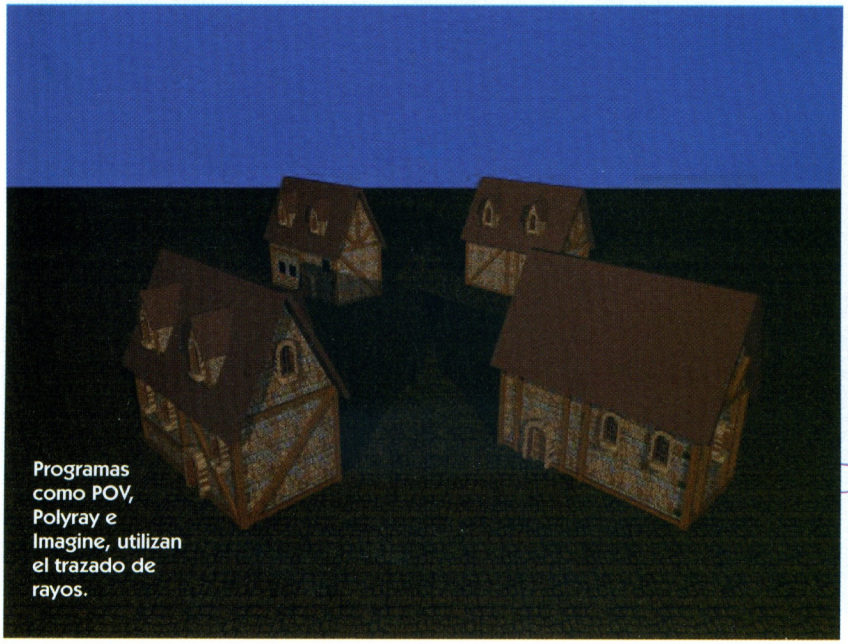
de POV el trazador de rayos más popular del mundo PC. Es por estas razones que POV ha sido el caballo de batalla fundamental del suplemento Rendermanía y por esta misma razón se le dedican estas páginas.

El trabajo con POV

Para crear una escena con cualquier software de generación de imagen de síntesis, ya empleamos trazado de rayos o cualquier otro método, habremos de suministrar al programa una descripción de la escena a renderizar en un formato comprensible. Habrá que indicar la posición espacial, forma y propiedades de los objetos, la colocación y orientación de la cámara y las fuentes de luz, y muchos otros detalles más. El fichero con la descripción necesaria puede crearse, básicamente, de dos maneras distintas: usando un programa de modelado o bien escribiendo directamente el texto con la descripción, empleando lo que se llama un lenguaje escénico.

El primer sistema es el que emplean paquetes como 3D Studio, Imagine, Caligari TrueSpace, Topas y otros. En cada uno de estos productos, la forma, el sistema de trabajo, y las opciones de cada modelador son diferentes, pero podemos destacar algunas características comunes: en todo programa de modelado existen una serie de opciones para construir objetos simples como esferas, cubos, conos, cilindros, etc., y también formas más complejas empleando métodos de extrusión, de revolución y otros. Luego, pueden realizarse diversas operaciones sobre estos objetos iniciales y obtener objetos aún más complejos. Después, podemos colocar cámara y luces, y ordenar la renderización de la escena. Muchas de estas operaciones pueden realizarse también con los lenguajes escénicos pero el empleo de un modelador es, en general, si esta bien diseñado, más rápido e intuitivo.

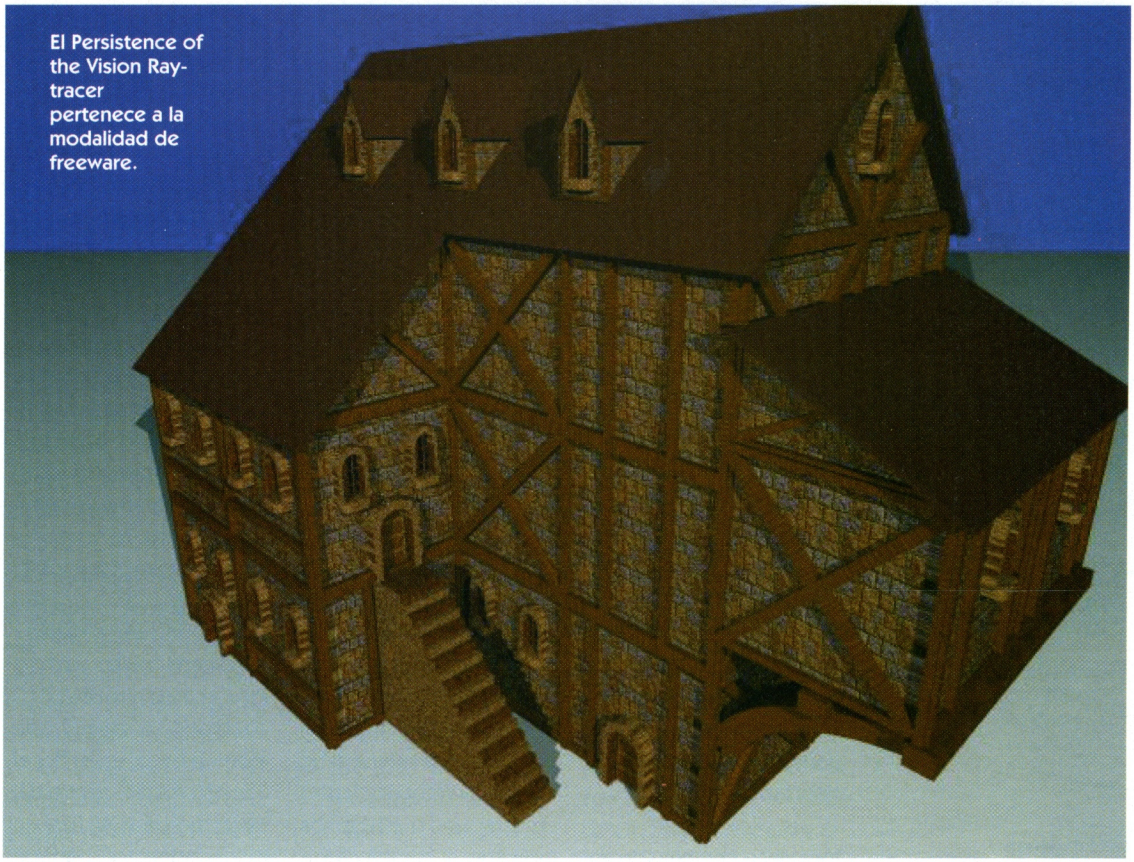
Veamos porqué: en un modelador podemos acceder a las distintas opciones empleando sistemas de ventanas o iconos, sin tener que escribir nada. Los objetos suelen visualizarse como mallas de hilos en una o más ventanas de trabajo con distintas vistas y, lo más importante, al ordenar una transformación cualquiera sobre un objeto; un desplazamiento, una escalación, una operación CSG, etc., veremos el resultado en las ventanas de trabajo, sin necesidad de ordenar un render.



Programas como POV, Polyray e Imagine, utilizan el trazado de rayos.

En POV, y en otros programas que carecen de modelador, el sistema es totalmente distinto. Usando un editor de texto cualquiera escribiremos un fichero que, empleando las sentencias permitidas por el lenguaje escénico, describirá la escena. Luego, invocando al programa, le daremos como entrada el fichero escénico escrito y le ordenaremos su proceso. En caso de que haya algún error de sintáxis en alguna sentencia del texto de entrada, el programa no mostrará nada y se limitará a indicar el punto del error en el fichero de texto. Entonces, deberemos corregir el error desde el editor de texto y volveremos a ordenar el render. Otra posibilidad es que el programa renderice la escena pero los resultados no sean los esperados, al haber cometido algún error lógico o espacial en la colocación de los objetos o en las operaciones entre estos. Cuando esto suceda, deberemos volver de nuevo al editor de texto para arreglar el problema. Este sistema es, como se ve, idéntico al que sigue un programador para compilar su trabajo y la mayor pega está en que los resultados de las operaciones no se visualizan "en tiempo real", como con un modelador. (Para visualizar cada cambio necesitaremos ordenar un render).

Pero no todas las ventajas están del lado de los mo-



El Persistence of the Vision Raytracer pertenece a la modalidad de freeware.

deladores. En primer lugar, algunas de las sentencias del lenguaje escénico de POV nos permitirán leer y emplear objetos creados con modeladores, con lo que, si disponemos de alguno, podremos aprovechar sus ventajas. Por otro lado, las sentencias de POV crean objetos cuya forma viene definida por fórmulas y no por polígonos, como suele ser el caso de todos los modeladores. Esto implica dos cosas: la primera, que los ficheros creados con un lenguaje escénico suelen ocupar mucho menos espacio que los archivos generados por los modeladores (incluso en el caso de que los modeladores realicen la grabación en algún formato comprimido propio). Pensemos, por ejemplo, en una esfera. Su definición tan sólo requerirá unas pocas palabras con un lenguaje escénico, pero cualquier modelador habrá de almacenar una lista de vértices y polígonos.

Otro efecto de esto es que una esfera definida por una fórmula es perfecta: por mucho que acerquemos la cámara a ella, seguiremos apreciando correctamente su esfericidad. En cambio, una esfera definida por polígonos tan sólo aparentará serlo si el número de polígonos empleado es lo bastante alto y la cámara no la enfoca a corta distancia. Finalmente, hay que añadir que la nueva versión de POV incluye importantes mejoras en el lenguaje escénico. Gracias a ellas, podremos conseguir escenas que impresionarán a cualquier usuario de un programa de modelado (a menos, claro está, que disponga de un paquete cuyo coste sobrepase las 400.000 mil pesetas o más). En todo caso, para quienes no lleguen a acostumbrarse al lenguaje escénico, la popularidad de POV ha propiciado la aparición de diversos modeladores que generan ficheros de escena directamente digeribles por POV. Estos modeladores representan los objetos de manera "poligonal" pero graban las escenas empleando el lenguaje escénico de POV. Esto origina una serie de problemas que parecen únicos de estas utilidades: no se representan bien las operaciones CSG, no pueden leerse archivos de POV, etc.

(Por ello, preferimos emplear sólo el lenguaje escénico. La librería Castlib ha sido creada "manualmente", sin emplear ningún modelador).

Además, como estos modeladores están estrechamente relacionados con el lenguaje de POV es recomendable aprender todo lo que se pueda a este respecto.

El uso de POV

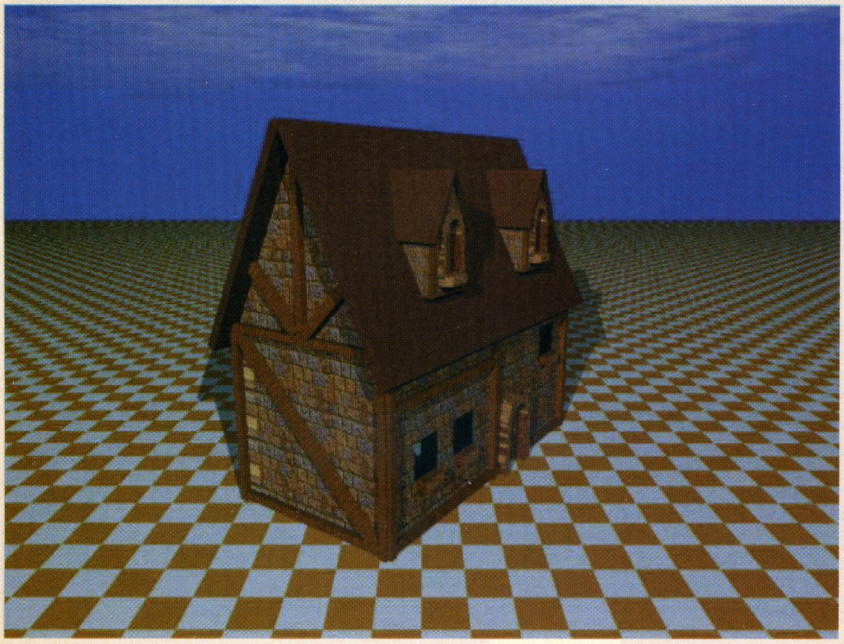
En las siguientes líneas vamos a analizar los pasos para instalar la nueva versión de POV. La versión cuya instalación vamos a comentar es la de PC bajo MS-DOS ya que es la más popular. Esta versión puede ejecutarse también desde una caja-MS DOS de Windows. (Temas que se trataron en el número 47 de PCmanía, con motivo de la publicación de POV, pero también nos parece adecuado insertarlos aquí).

POV viene comprimido en un fichero llamado Povms-dos.zip. Deberemos guardar este fichero en un directorio temporal y, desde allí, procederemos a descomprimirlo usando el programa Pkunzip. Hecho esto, aparecerá ante nosotros el verdadero paquete de instalación: Install.exe. Este programa requiere 600 Kbytes de memoria convencional y es autoexplicativo (aunque en inglés, claro). Al ejecutarlo, el programa solicitará que especifiquemos un disco y un directorio donde guardar el paquete a instalar. Hecho esto, se procederá a realizar la instalación, creándose una serie de subdirectorios dentro del directorio que hayamos indicado al programa de instalación. Al finalizar el proceso, aparece un texto del coordinador del Povteam, explicando algunos detalles sobre la versión.

Town.inc

Con este suplemento incluimos la versión 1.8 de castlib compuesta por un único fichero llamado town.inc. Además, se incluyen los ficheros de escena llamados escenaxx.pov, los ejemplos llamados ejempxx y los ficheros bat.

(Las xx indican el número de escena o ejemplo). Lo ideal es crear un nuevo subdirectorio -al que desde ahora nos referiremos por el nombre: media- dentro del directorio de POV. Será en este subdirectorio donde ejecutemos todos los ejemplos de uso de la librería.



Las imágenes se crean por defecto con +Q9.

Ejecutar POV bajo DOS y Windows

Como sucedía en versiones anteriores, el fichero autoexec.bat deberá indicar un path a povray.exe si deseamos ejecutar el programa desde cualquier directorio. También, deberemos especificar el path a los nuevos ficheros include usando la orden "L" de la línea de comandos. Una vez impartida la orden a POV, el programa funcionará igual que siempre, aunque con algunos pequeños cambios. En primer lugar, cuando la imagen termine de generarse, POV

no saldrá al DOS, sino que nos permitirá observar distintas pantallas informativas con estadísticas, opciones usadas en la generación, e información del programa. Podremos cambiar de una pantalla informativa a otra, usando las teclas de flecha y salir de POV con Return. En caso de fallo, POV nos dará la posibilidad de examinar dos pantallas adicionales "Error" y "Debug".

Por otro lado, como se ha dicho antes, la versión MS-DOS de POV puede funcionar desde una caja DOS de Windows. Teóricamente, podemos lanzar varios renders simultaneamente empleando varias cajas pero, como podéis imaginar, esto no es demasiado práctico. Por un lado, una única CPU deberá repartir su tiempo entre dos o más procesos de render, con lo cual no ganaremos tiempo con respecto a una lista de renders secuenciales (ordenados desde un fichero bat). Otro posible inconveniente está en la visualización.

Aunque la generación de la imagen sea correcta, es posible que haya fallos de visualización al cambiar de una "caja-Dos" a otra.

Documentación

POV siempre ha contado con una excelente documentación en inglés que, en la nueva versión, ha sido ampliada y apoyada por la inclusión del programa Povhelp, con el que podremos acceder a cualquier tema, desplazándonos con las teclas de flecha sobre los títulos y pulsando Enter para acceder a los temas.

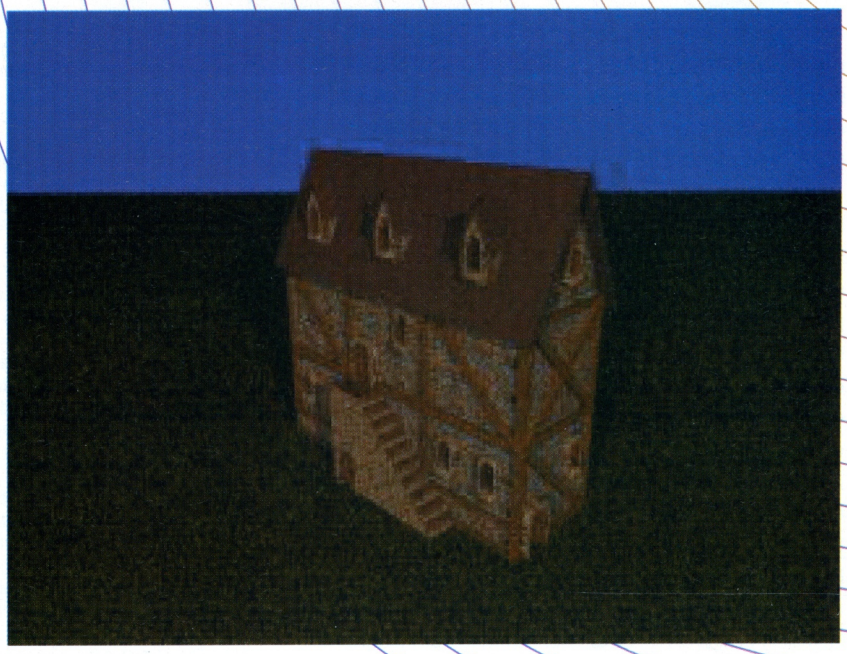
También, para quienes lo prefieran así, resulta posible obtener un fichero ASCII con el manual, usando la siguiente línea:

phe2txt -ipovhelp.phe -f

Ficheros .inc y escenas de ejemplo

Otra valiosa fuente de información para el povmaníaco novato está en los muchos ejemplos que incluye POV. Estos ejemplos son ficheros ASCII con una descripción de una escena tridimensional formulada usando las sentencias permitidas por el lenguaje escénico de POV, como antes se dijo. Los archivos de POV, usan siempre las extensiones .pov o .inc para indicar que son ficheros escénicos de este Ray-tracer. Un escena de POV puede usar uno o muchos de estos archivos. Usualmente en cada proyecto sólo habrá un archivo con la extensión .pov. Este fichero será el principal y el que cargará a los archivos .inc con órdenes include. Como ya supondréis, cualquier proyecto descrito con estos archivos debe definir una cámara que "retrate" la escena, luces que la iluminen, y objetos diversos. Si alguno de estos tres tipos de elementos no se incluye en el proyecto, el resultado será una pantalla en negro.

En su versión 3.0, el usuario puede consultar diversos tipos de ejemplos: en el directorio Pov3demo veremos varios subdirectorios, cada uno de los cuales incluye varios ejemplos del tema a que hace referencia el nombre del subdirectorio: atmos (efectos atmosfé-



Escena a 320x200 pixels de resolución.

ricos), camera (ejemplos de uso de la cámara), objects (uso de los objetos simples que pueden crearse con POV), lights (luces), etc.

La nueva versión también conserva los ejemplos de versiones anteriores en el directorio povscn. Estos ejemplos estan divididos en varios subdirectorios, según su complejidad. Por último, en el directorio texsamps podremos generar escenas dedicadas a mostrarnos las librería de texturas de POV. La mayoría de estas texturas han sido creadas por artistas de POV (muchas son de Mike Miller) y están clasificadas en subdirectorios, woods (maderas), metals (metales), stones (piedras), skies (cielos) y glasses (cristales). Todas estas magníficas texturas han sido creadas usando sentencias del lenguaje de POV y el lector puede examinar las definiciones en el directorio llamado include.

La manera ideal de estudiar el lenguaje escénico es realizar experimentos con los ficheros del directorio pov3demo, ya que son los más recientes. En particular con los ficheros de objects, camera y lights. Podremos probar las órdenes de la línea de comandos o probar cambios en las sentencias de los ficheros y aprender de ello. (También conviene hacer copias de los ficheros originales). Además, los ficheros de ejemplo pueden usarse como punto de partida para crear nuestras propias escenas.

Comandos de la línea de parámetros

Estos comandos controlan la calidad, forma, tamaño, y modo en que se generan las imágenes y especifican el nombre del fichero escénico de entrada y el del archivo gráfico de salida, en caso de que vayamos a crearlo. Los comandos de la línea de órdenes pueden ser dispuestos en cualquier orden y van precedidos de los simbolos + o -. Por norma general, el simbolo + suele activar el parámetro que le sigue y el - lo desactiva.

Cuando se está trabajando en un proyecto puede ser útil renderizar escenas distintas o emplear diversos parámetros. Por esta razón, y para no tener que estar trasteando continuamente con las líneas de órdenes, pueden usarse diversas alternativas. La primera es tener uno o más ficheros bat para invocar a POV con los comandos y órdenes pertinentes.

Otra posibilidad es crear ficheros ini, donde escribiremos una lista de opciones por defecto. Los ficheros ini admiten los mismos comandos que las líneas de órdenes y podemos invocarlos desde la línea de órdenes, y mezclar su uso con comandos de línea. Veamos por ejemplo: POVRAY RES800 +Iprueba.pov +oprueba.tga.

La primera palabra de la línea es el nombre del ejecutable de POV. Sigue el nombre del fichero ini, y luego dos comandos de lí-

+Hn Resolución vertical de la imagen a generar. N es la altura en pixels.
+Wn Resolución horizontal de la imagen a generar. N es la anchura en pixels.
+D Activa la visualización de la escena mientras se va generando.
-D La desactiva.
+Dn Activa la visualización en el modo gráfico número n.
Lista de tarjetas soportadas por +D +D0 Autodetección de (S)VGA type (puesta por defecto) +D1 Standard VGA 320x200 +D2 Standard VGA 360 x 480 +D3 Tseng Labs 3000 SVGA 640x480 +D4 Tseng Labs 4000 SVGA +D5 AT&T VDC600 SVGA 640x400 +D6 Oak Technologies SVGA 640x480 +D7 Video 7 SVGA 640x480 +D8 Video 7 Vega (Cirrus) VGA 360x480 +D9 Paradise SVGA 640x480 +DA Ahead Systems Ver. A SVGA 640x480 +DB Ahead Systems Ver. B SVGA 640x480 +DC Chips & Technologies SVGA 640x480 +DD ATI SGVA 640x480 +DE Everex SVGA 640x480 +DF Trident SVGA 640x480 +DG VESA Standard SVGA Adapter +DH ATI XL display card +DI Diamond Computer Systems SpeedSTAR 24X +Dnp Idem. que orden anterior pero usando el modo de paleta número p.
Lista de tipos de paleta soportados +D?3 Paleta 332 (la usada por defecto). Esta compuesta por 256 colores con 3 bits para el componente rojo, 3 para el verde y 2 para el azul.
+D?0 Paleta de 256 colores que usa el sistema de color HSV.
+D?G Usa una paleta de 64 tonalidades de gris.
+D?H Opción HiColor (32000 colores con 5 bits por componente).
+D?T Opción Truecolor (color real). 24 bits bajo Vesa.
Cuando solo nos interesa visualizar una pequeña zona de la imagen (porque solo se esperan cambios en ella), utilizamos los siguientes 4 comandos: +SCn La columna n sera la inicial del recuadro parcial a renderizar.
+ECn Para el mismo recuadro, n sera la columna final.
+SRn Columna inicial del recuadro.
+ERn Columna final del recuadro.
+X Permite interrumpir la generación de la imagen pulsando una tecla.
+C Permite reanudar el calculo de una imagen interrumpida con +X.
+P Al finalizar el cálculo de la imagen, POV esperara a que se pulse una tecla para salir. Usamos +p para ver la imagen cuando ha concluido el calculo. +P se usa junto a +D.
+V Cuando, por la razón que sea, decidimos desconectar el display de la imagen (-D), puede ser conveniente saber que línea esta calculando POV en cada momento. Con +V activaremos esto. En caso de que se active la visualización (+D), deberemos usar -V.
+Iname Junto a +I, se indica el nombre del fichero escénico de entrada.
+Oname Junto a +O, se indica el nombre del archivo gráfico de salida.
+Lpath Por defecto POV sólo examina el directorio en curso para buscar los ficheros include que solicita el fichero .pov que se va a calcular. Con la opción +L podremos indicar una ruta a un directorio donde hayamos almacenado ficheros de inclusión. Es posible emplear varias ordenes +L.
+Qx El valor x indica la calidad del render. Valores de 0 o 1 usaran sólo iluminación ambiental, 2 y 3 usarán luz difusa, 4 renderiza sombras y 5 aplica luces zonales, 6 y 7 calculan texturas, el valor 8 calculará rayos reflejados, refractados y transmitidos y a partir de 9 se calcularan halos. El valor por defecto es 9.
+QR Se calcula la radiosidad de la escena.
+Bx Normalmente, cada vez que concluye el cálculo de una línea, POV la graba en el archivo gráfico de salida. Si queremos minimizar el número de grabaciones emplearemos este comando cuyo valor x es el número de Kbytes reservados para el buffer de grabación.
+A Activa el antialias al valor por defecto (0.3). El antialias se emplea para suavizar la apariencia "dentada" que tienen en la escena las líneas diagonales. Por defecto el antialias esta desactivado (-A).
+Ax Cambia el valor para el antialising.
+Kx Usada para pasar un valor x en flotante a la variable clock que es empleada por POV para hacer animaciones.
+MVx Emplea la sintaxis de la versión x de POV (para ejecutar escenas antiguas).
+SPx Usa un efecto de mosaico que nos permite hacernos una idea del aspecto general de la imagen antes de que esta se calcule por entero. Si por ejemplo x=16, se calcularan y visualizaran puntos de 16x16 pixels. Luego la representación pasará a puntos de 8x8 pixels, y asi hasta llegar al cálculo pixel a pixel.

nea de órdenes que indican, primero, el nombre del fichero escénico a procesar y, después, el nombre del archivo gráfico a producir. En los ficheros ini las opciones pueden especificarse también con ciertas palabras en vez de con comandos de línea pero, al ser un tema algo redundante, no las trataremos en esta ocasión. Nos limitaremos a los comandos de línea más importantes, los cuales pueden verse en la **figura 1**. Ahora, veamos varios ejemplos con estos comandos.

Situándonos dentro del directorio media probemos la siguiente línea:

povray +iejemp1.pov +oejemp1.tga+d0-v+w320 +h200-+lc:\povray3\include.

El fichero de ejemplo ejemp1.pov será procesado y se creará un fichero tga con el mismo nombre. POV autodetectará la tarjeta disponible y visualizará la escena a una resolución de 320x200 pixels, sin antialias. Los ficheros include necesarios para la generación se buscaran en la ruta indicada por el comando +L.

Si ahora probamos nuevamente la línea precedente añadiendo +Q4, obtendremos un resultado mucho más pobre a cambio de una mayor velocidad de generación.

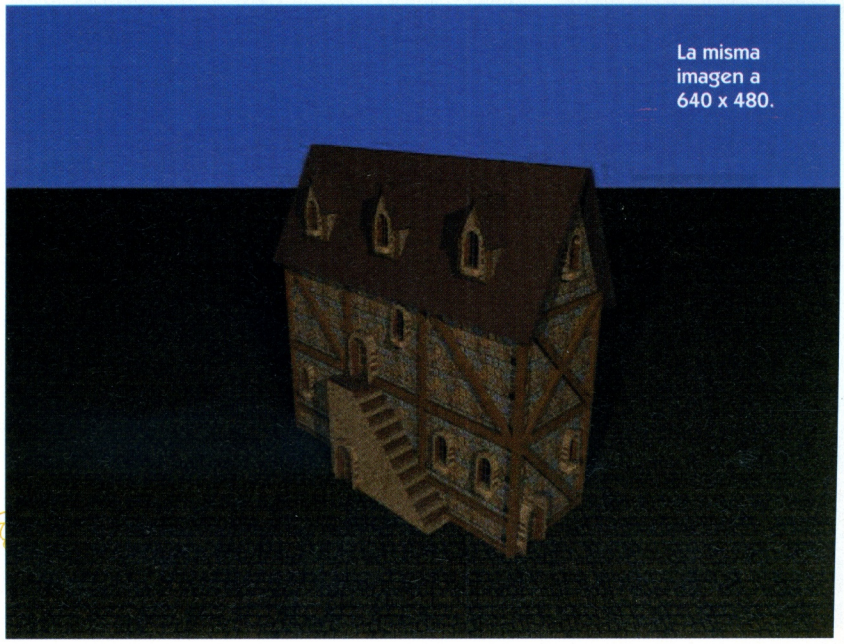
Consejos sobre comandos de línea

Como veremos más adelante, antes de obtener el resultado final buscado en una escena hay que realizar bastantes pruebas, hay que construir cada objeto, ponerlo en su posición adecuada, elegir las texturas, la iluminación adecuada, etc. Una buena resolución para una escena final puede ser 800x600 pixels, pero resulta absurdo calcular las pruebas intermedias con estos valores. Normalmente 320x200 o incluso 160x100 pixels, suelen ser cifras más prácticas para 'W' y 'H' durante los renders de prueba. Además de esto, puede usarse la opción +SP con valores de 8 o 4.

También, conviene utilizar el buffer poniendo +B con valores de 64 o más. De este modo, ahorraremos algo de tiempo y nuestros oídos sufrirán menos con el ronroneo del disco duro. Rebajar la calidad del render no es tan buena idea como en principio pudiera parecer. La pérdida de calidad puede despistar bastante y, a la larga, hacernos perder el tiempo. Es preferible hacer pruebas parciales a calidad total y desactivar los elementos de la escena que no vayan a comprobarse en las pruebas. Otra buena idea es desactivar el antialias (-A). El cálculo del antialias siempre requiere tiempo adicional y sólo tiene sentido activarlo en el render final.

Por último es conveniente conservar la posibilidad de poder interrumpir siempre la generación de la imagen a golpe de tecla (+X), ya que más de una vez percibiremos fallos garrafales mucho antes de que la escena se haya generado por completo. Si la interrupción se efectua en medio de una imagen importante, para continuar con ella posteriormente bastará con añadir +C a su línea de ordenes. Los comandos SC, EC, SR y ER, rara vez se emplean para nada.

En cuanto al comando 'D', si se tiene una tarjeta mínimamente moderna, lo más conveniente es usar +DGH o +DGT. No visualizar la imagen (-D) apenas ahorrará tiempo y perderemos la posibilidad de darnos cuenta pronto de si hay un error. POV, por defecto siempre genera un archivo tga. Si no especificamos un nombre para él, éste será output.tga. Si queremos evitar su creación podemos hacerlo con -F pero..., ¿para qué hacer esto? Siempre puede ser útil organizar y guardar cada prueba. Podemos examinarlas usando algún visor de ficheros gráficos como alchemy e incluso convertir los ficheros tga a jpg, si el espacio ocupado se dispara. Más tarde volveremos nuevamente al aspecto "tiempo", ya que es un detalle crucial en el trabajo con un trazador de rayos.



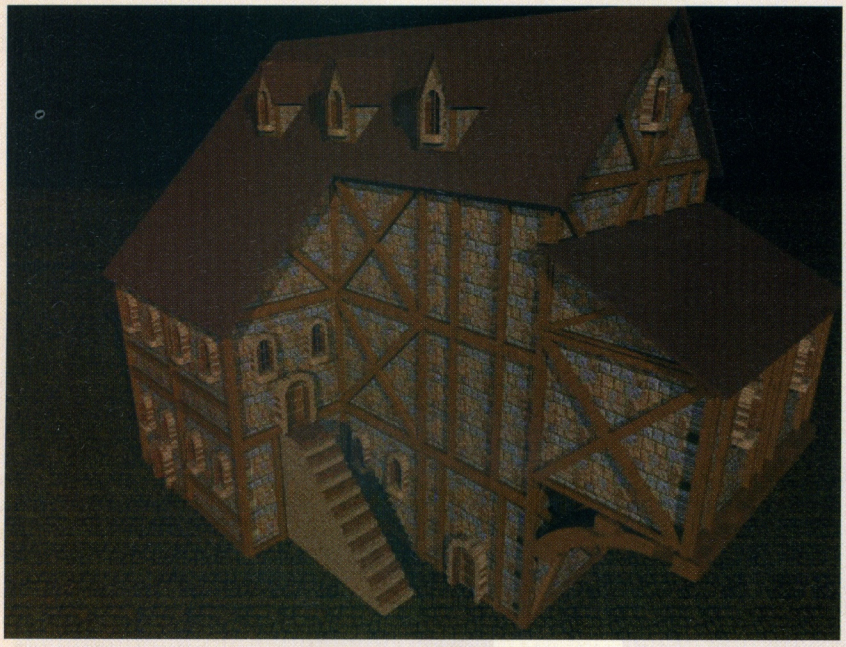
La misma imagen a 640 x 480.

El universo tridim

Podemos pensar en el universo virtual de POV como un espacio que se extiende infinitamente en todas direcciones. En este universo tridimensional todos los elementos: objetos, cámaras y luces, han de tener una colocación espacial que se declarará con tres valores de coordenadas para los ejes X,Y y Z.

Si recordamos algunas nociones básicas de geometría, sabremos que los ejes eran conceptos abstractos utilizados para situar puntos u objetos en el espacio. El punto donde se cruzaban los ejes era llamado "origen de coordenadas" y asumiremos que es el centro del universo del Pov-espacio. Naturalmente, las coordenadas X,Y y Z de este origen son <0,0,0>. Si imaginamos que este punto corresponde al centro de nuestra pantalla de ordenador, podremos visualizar el eje X como una línea invisible que recorre horizontalmente nuestro monitor, siendo el eje Y una linea similar que cruza verticalmente el centro de la pantalla y el eje Z otra línea que corta perpendicularmente el plano de la misma, entrando o saliendo de ella. De esta manera, y siempre asumiendo que el centro de la pantalla corresponde al origen de coordenadas, las posiciones del eje X, situadas a la derecha del mencionado origen, son positivas y las situadas a la izquierda negativas. Las posiciones del eje Y que quedan por encima del origen son positivas, y las inferiores negativas. En cuanto al eje Z, asumiremos que la parte del mismo que penetra en el monitor es positiva y la opuesta negativa. Todos los valores crecen positiva o negativamente conforme los objetos se alejan del centro de coordenadas.

Esta disposición en la colocación y en el sentido de los ejes es totalmente arbitraria y otros programas emplean convenios diferentes. Por lo que respecta a las unidades utilizadas en Pov, podemos suponer que están en metros, en micras, en años-luz, o en cualquier otra medida mientras trabajemos de manera consistente y respetemos las relaciones de tamaño entre objetos.



Escena iluminada con una fuente de color blanco.

Objetos virtuales

El lenguaje escénico de Povray nos permite definir un universo tridimensional poblado de objetos virtuales. Estos objetos pueden ser muy simples o extremadamente complejos, dependiendo de lo que nuestra imaginación, habilidad

ensional del POV

artística, o paciencia, nos permitan. Podemos crear modelos complejos a partir de muchos objetos simples, situarlos en el espacio, iluminarlos, y enfocarlos con la cámara desde cualquier posición. Además, habremos de definir la apariencia de estos objetos. En el mundo real las superficies pueden ser rugosas o lisas, mostrar los caprichosos dibujos del mármol o las líneas onduladas de la madera, exhibir brillos metálicos, etc.

En POV, todas estas propiedades se definen en las texturas que se asignan a los objetos. Elegir y aplicar correctamente las texturas en los objetos es algo tan vital como la propia forma de estos. La definición de una textura puede ser algo muy sencillo o bastante complejo pero no nos asustemos: POV incorpora una completa librería de texturas con apariencias de maderas, mármoles, granitos, cielos, etc.

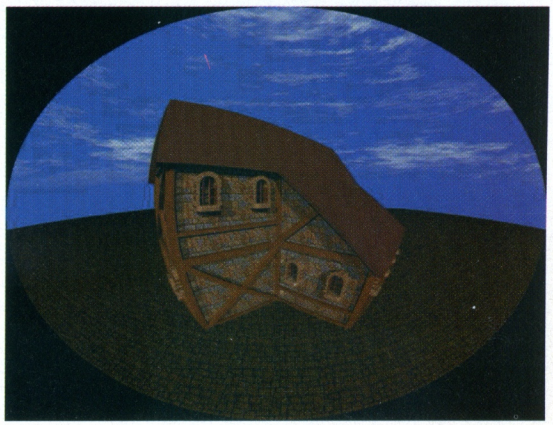
Veamos el formato general que sigue la definición de un objeto: tipo_objeto{posicion_tamaño_y_orientación transformaciones textura{propiedades transformaciones} transformaciones} Las llaves definen el "ámbito" de las sentencias que definen al objeto. O sea que las sentencias propias de la definición de la textura deben estar encerradas entre las llaves correspondientes de la misma y lo mismo ocurre con las transformaciones espaciales del objeto. Si hay alguna llave de más o de menos, o si la colocación de las instrucciones no corresponde a la que debe ser en el apartado correspondiente; obtendremos un error y la escena no se creará o cuanto menos no se creará correctamente. En cuanto a la palabra "transformaciones", se refiere a las transformaciones espaciales: traslación, rotación y escalación y su uso es siempre opcional. También, es muy importante el sitio donde se efectue cada transformación. Por ejemplo, en el formato general antes descrito la palabra transformación figura tres veces: en la primera, la transformación sólo afectaría al objeto en sí pero no a la textura ya que la definición de ésta sigue después. Por el contrario, la siguiente vez la transformación afecta sólo a la colocación, escala y orientación de la textura y no al objeto, que no sufre estos cambios. Esto es así porque la palabra está delimitada por las llaves de la definición de la textura.

Por último: la tercera transformación afectará por igual a la forma y a la textura del objeto.

Fuentes de luz

El universo espacial de POV está, por defecto, a oscuras. Para poder ver algo tendremos que crear, al menos, una fuente de luz. Las fuentes de luz son declaradas y tratadas como objetos, con la salvedad de que son puntos infinitamente pequeños: no pueden, pues, ser vistas ya que carecen de tamaño y forma. Esto es así por cuestiones técnicas, ya que todos los rayos luminosos han de salir del mismo punto, lo que a su vez provoca los problemas mencionados al principio con las sombras.

La verdad es que POV emplea diversos trucos para paliar estos problemas. En cuanto a los tipos de fuente, la más utilizada y sencilla -y a la vez la que menos cálculos requiere- es la fuente de tipo omnidireccional. Ésta derrama su luz en todas direcciones, tal



Una vista de nuestra casa desde una cámara diferente de POV.

Una cámara de POV.

como lo hace, por ejemplo, el sol. Veamos cómo crear una fuente de este tipo:

light_source { <30, 20, -50> color White}

Como vemos los datos de la fuente de luz quedan encerrados entre llaves, de manera similar a como sucedía en los objetos virtuales. Las llaves que delimitan la definición de esta fuente engloban dos datos: un vector utilizado para indicar la posición espacial de la fuente en el universo virtual, y el color de la misma. Los vectores son profusamente utilizados en POV y más tarde los trataremos con mayor detalle. En este momento, nos bastará con saber que el vector del ejemplo señala la posición X, Y y Z de la fuente. Después del vector, sigue la palabra reservada "color", la cual se emplea para especificar aquello que su nombre indica.

Una de ellas es precisamente la empleada en el ejemplo anterior: la palabra White (blanco) es en realidad un identificador o macro definida con #declare en el fichero de inclusión colors.inc. Si incluimos en nuestro fichero escénico la sentencia precisa (#include) para cargar este fichero inc, podremos referenciar cualquier identificador incluido en el mismo. El funcionamiento de esta especie de macro es muy sencillo: al hallar el nombre del identificador, Pov lo sustituye por el texto que, en la declaración, sigue al identificador. Esta potente idea, que examinaremos en profundidad en la sentencia #declare, es la base de gran parte de la potencia del lenguaje de POV. El lector puede examinar el fichero colors.inc (en el directorio include) para comprobar la lista de colores "declarados" por POV.

(Es muy importante recordar que POV distingue entre mayúsculas y minúsculas. Cuando usemos una sentencia o un identificador tendremos que escribirlo tal como se indique. En el ejemplo anterior white o WHITE no serían reconocidas por POV y obtendríamos un error).

En el fichero colors.inc viene definido un buen número de colores pero, teniendo en cuenta que muchas tarjetas pueden mostrar hasta 16 millones de colores, es obvio que necesitamos un método alternativo para especificar el color de la fuente de luz o de los objetos.

El color

El sistema RGB utiliza una mezcla de tres colores primarios para representar cualquier color. Es el sistema utilizado por los televisores a color y el empleado también por POV. Por tanto, deberemos definir cualquier color utilizando intensidades de red (rojo), green (verde) y blue (azul). POV emplea varias sintaxis para especificar colores. Una de ellas, algo antigua ya, es: color red 1 green 0 blue 0.

Los valores en formato flotante que siguen a cada color primario indican la fuerza del mismo y pueden oscilar desde 0 (nada) a 1 (máxima intensidad).

El ejemplo anterior es, evidentemente, la definición del rojo puro y, como los otros dos componentes están a cero, podemos escribir tambien: color red 1.

Pero esta sintaxis es algo engorrosa y por ello la siguiente línea también es válida: color rgb <1, 0, 0>.

De hecho, también podemos precindir tranquilamente de la palabra color y limitarnos a: rgb <1,0,0>.

Además, por una propiedad de POV llamada "promoción de operadores", es válida una línea como: rgb 1.

...que es la definición del color blanco. Ahora pasemos a otros detalles.

Realmente, la especificación de un color, aunque puede limitarse a tres valores, puede emplear hasta cinco componentes. El cuarto es llamado filter (filtro) y se usa para indicar la cantidad de transparencia de un material.

Un valor de cero en este componente señala una opacidad total y un valor de 0 una transparencia total. Veamos la definición de color usada en el fichero glass.inc para crear una textura empleada para las botellas de vino: color rgbf <0.4, 0.72, 0.4, 0.6>.

El cuarto valor equivale a una transparencia del 60%. Como podemos suponer, aún sin efectuar ningun render, el color del material será más bien verde, ya que tiene un fuerte valor en este componente. En cuanto al quinto componente, indica la cantidad de luz no filtrada que se transmite a traves de una superficie. Ejemplos de este tipo de transparencia (que no existía antes de la versión 3.0) pueden observarse en cortinas o telas finas. Este componente es llamado transmit y podemos agregar una 't' a la palabra rgb para aludirlo. Así usando rgbt, el cuarto valor corresponderia a este componente y en rgbft al quinto.

Por último, hemos de tener en cuenta que, al igual que sucede con el mundo real, el color de un objeto depende también de la fuente de luz,

sobre todo si la superficie del objeto permite un alto índice de reflexión.

La cámara: cómo situarla y orientarla

Al igual que sucede con las fuentes de luz, las cámaras son objetos invisibles. Cada escena debe incorporar una definición de la cámara que va a sacar la "fotografía". En dicha definición le indicaremos al programa donde situar nuestra cámara, hacia donde va a mirar, si va a tener zoom o no, etc. Así, como vemos, en la colocación de las cámaras tiene lugar un proceso muy similar al seguido por un fotógrafo o un director de cine.

(Sobre todo porque existe la posibilidad de crear películas con las imágenes generadas con POV).

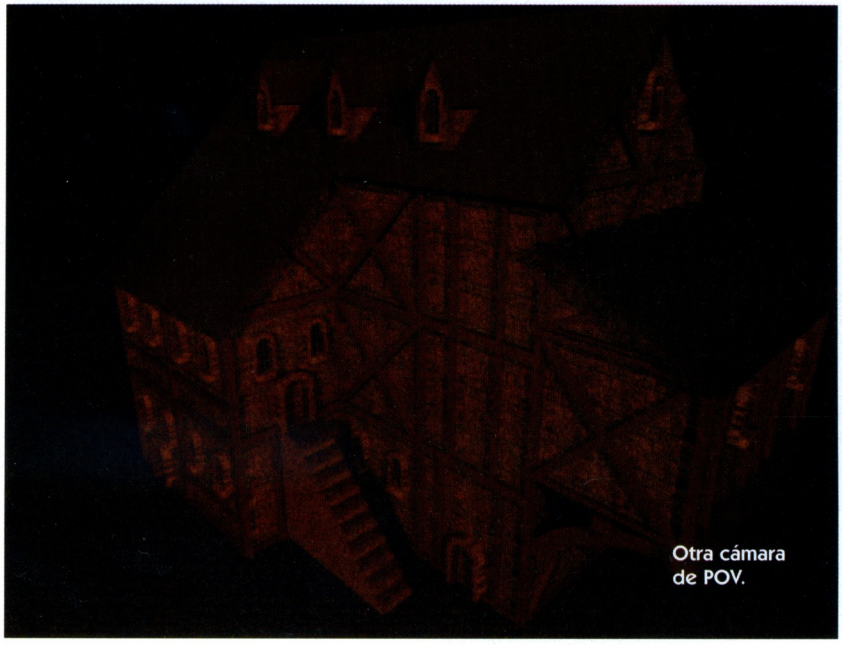
Aquí tenemos una posible definición para la cámara de una escena: camera { location <0, 145, -120.0> direction <0.0, 0.0, 1.5> up <0.0, 1.0, 0.0> right <1.33,0.0, 0.0> look_at <3, 10, 0.0 > }.

La colocación de la cámara en el espacio se logra con la instruccion Location. Esta sentencia utiliza tres números que indican la colocación de nuestra cámara, con respecto al origen de coordenadas. En el ejemplo anterior, nuestra cámara estará situada a 120 unidades fuera del monitor (si damos por válido el ejemplo que se comentó al explicar el sistema de coordenadas, donde se dijo que el valor 0 de Z estaba situado en el plano de la pantalla. Según este mismo ejemplo, si el valor 120 fuera positivo, la posición caería dentro del monitor y la escena se enfocaría desde el lado opuesto). Además, la cámara se posicionará a 145 unidades por encima del origen (por el valor del eje Y).

Look_at es, aparte de location, la instrucción más importante dentro del grupo de las que componen la declaración de la cámara. Especifica la posicion hacia la que esta va a mirar, forzándola a girarse para enfocar la posición espacial indicada en el vector que sigue a look_at. En el ejemplo anterior, la cámara enfocaría un punto colocado a la derecha y un poco por encima del origen de coordenadas (ya que Z=0). La sentencia direction indica la dirección inicial de enfoque de la cámara, dirección que, luego, puede ser cambiada por las sentencias look_at,, rotate u otras. La línea de enfoque de la cámara sigue el camino indicado por el vector de dirección (normalmente <0,0,1>). La longitud de este vector, 1.5 en el ejemplo, también afecta la distancia de la ventana de visión a la cámara. Una valor grande da como resultado un campo de visión corto, ya que el plano de visión quedará muy próximo a la cámara.

La instrucciones up y right definen respectivamente la dirección vertical y horizontal del plano de visión. También afectan al aspect ratio de la imagen a "fotografiar". En el ejemplo, up tiene el valor 1 en el eje Y vertical y right el valor 1.33 en el eje X. Con ello, se indica que la cámara no tiene ninguna inclinación con respecto al plano X,Z, tal como sucedería si estuviese apoyada en un trípode quedando paralela al suelo. En cuanto al valor 1.33, normalmente las resoluciones para las imágenes a generar suelen ser de 640x480, 800x600, 1024x768, etc. Estos valores tienen una relación de 1.33 (640/480, 800/600, etc). Por ello, colocando esta relación en right se efectúa una compensación, de modo que los objetos no quedan distorsionados. En las figuras podemos ver un ejemplo, cuya longitud es idéntica en los ejes X e Y.

Teóricamente, usando tan sólo las sentencias location, direction, up y right, resulta posible colocar y orientar la cámara de cualquier forma pero, en la práctica, esto es muy difícil, y por ello, contamos con la ayuda de look_at.



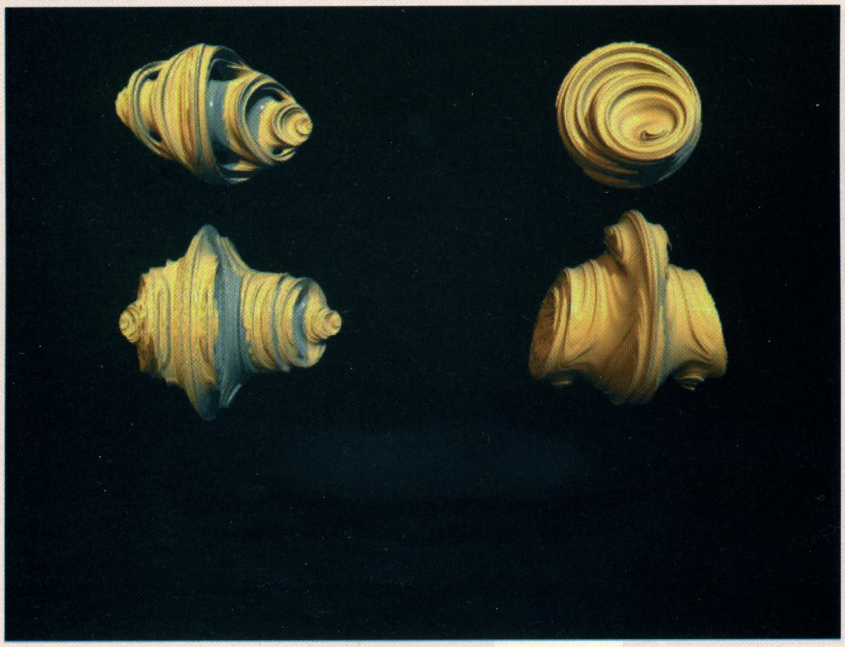
Otra cámara de POV.

Componentes del

Los componentes del lenguaje escénico de POV son identificadores, sentencias, vectores, cadenas, símbolos especiales y comentarios. Todos estos elementos pueden usarse de una forma bastante libre para construir un fichero escénico.

No hay tabulaciones determinadas ni es preciso seguir un orden concreto en la colocación de los elementos, si bien es adecuado atenerse a algun tipo de formato por criterios de claridad, sobre todo si se va a desarrollar un proyecto medianamente complejo. En town.inc se advertirá un estilo determinado que el lector puede sustituir tranquilamente por otro con el que se encuentre más cómodo. Veamos ahora algo sobre los componentes del lenguaje.

Primitivas complejas de POV: objetos fractales.



Instrucciones e identificadores

El lenguaje escénico del POV consta de una larga lista de instrucciones y palabras reservadas utilizadas como parámetros en dichas sentencias. El lector puede observar esta lista en la figura sin sentir escalofríos; en principio basta con saber emplear bien unas cuantas sentencias y reglas para poder realizar escenas bastante vistosas. La primera regla a recordar es sencilla: POV nos permite crear palabras para declarar objetos, colores, texturas, etc., y no podremos utilizar para ello ninguno de los nombres de la figura, dado que ya que están reservados para un uso determinado. Por ello, estos nombres, ya sean de instrucciones o de parámetros, reciben también el nombre de palabras reservadas. Por otra parte, las posibles palabras definibles por el usuario son llamadas identificadores. Los identificadores pueden tener hasta cuarenta caracteres de largo y se escriben con cualquier caracter alfabético o numérico, además de con el caracter '_'. (Aunque el primer caracter del identificador deberá ser alfabético y no numérico). Finalmente, debemos recordar que no se permiten espacios dentro del nombre del identificador. Los identificadores se crean con #declare. Por ejemplo: #declare Chrome_Metal = texture { Chrome_Texture }.

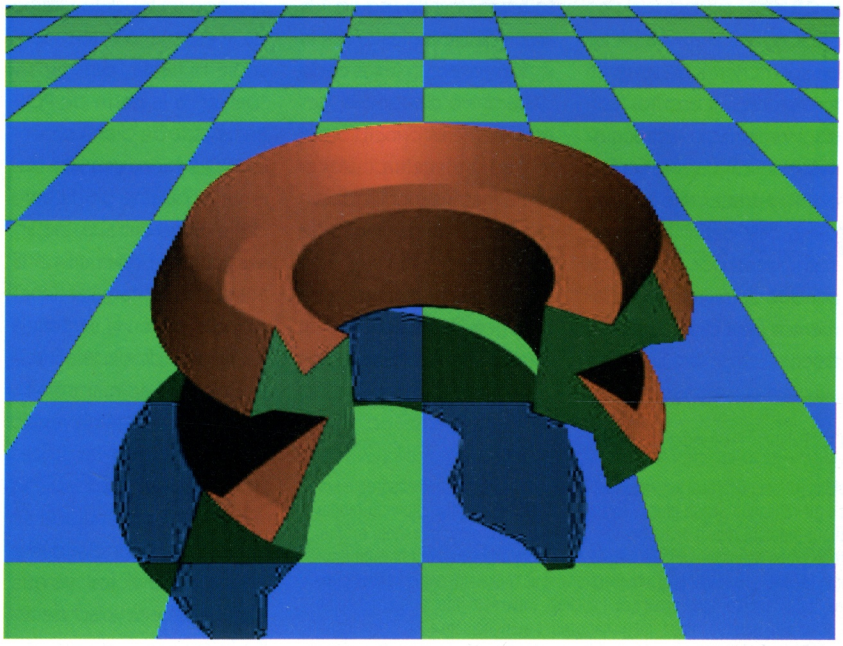
Esta línea crea la textura Chrome_Metal.

Comentarios

Todos los lenguajes de programación tienen la posibi-

lenguaje escénico

lidad de incluir comentarios a fin de facilitar la comprensión del programa. Esta posibilidad es especialmente útil cuando un programador debe estudiar listados ajenos. Pero incluso aunque el listado sea propio puede ocurrir que, con el paso del tiempo, ciertas líneas especialmente complejas se hagan difíciles de entender. Esta situación, también puede darse con el lenguaje escénico y, por ello, el pov-team ha incluido la posibilidad de introducir comentarios. Un comentario puede constar de una o más líneas de texto delimitadas por separadores y, su contenido, no tiene ningun efecto para la creación de la escena. Tan sólo servirá para hacer más comprensible al lector el significado del listado. Los comentarios del lenguaje escénico del POV utilizan las mismas reglas que los del C y C++. O sea una doble barra para los comentarios de una sola línea y los caracteres /* y */ para los de una o más líneas. Veamos unos ejemplos: // Esta línea es un comentario y el POV la ignora /* Esta línea es ignorada por el "intérprete" que efectua el proceso de parsing del POV. De hecho, todo es ignorado hasta el final de esta línea */. También, es posible insertar un comentario después de un trozo de línea "útil" del siguiente modo: object {cualquier_cosa} //El POV sólo ignora esta frase.



Primitivas complejas de POV: objeto lathe.

Los comentarios también son muy útiles cuando se desea eliminar temporalmente un trozo de escena consistente en una o más líneas. Es mucho más fácil recuadrar el trozo con los símbolos de marca de comentario que simplemente borrarlo y volverlo a escribir después.

Valores en formato flotante

El manual de POV define un vector como un conjunto de valores en flotante.

(El formato flotante es empleado para almacenar números con un componente real y otro entero). Un vector puede tener de 2 a 5 componentes separados por comas y delimitados por los caracteres '<' y '>'. Los vectores pueden estar compuestos por expresiones numéricas, identificadores o funciones que retornan valores. Ejemplos válidos de uso de vectores son:

translate<Col2 ,Row1 ,Dist>

scale <1 ,2.1 ,-25.4578>

location<posX,posY+15,-20>

translate <años_luzX/densidad_estelar ,años_ luzY+rand(a) +12 ,100>.

Naturalmente, el número de componentes de un vector dependerá de la palabra que preceda a dicho

vector. En rgbft serán 5, en una sentencia location se precisaran 3, etc.

"Parsing" y ficheros include

En POV, los archivos include son ficheros de texto en formato ASCII escritos con declaraciones o instrucciones del lenguaje escénico del Ray-tracer. No se emplean para definir imágenes completas por sí mismos, aunque sí incorporan elementos para ellas. Su filosofía consiste en incluir definiciones (declaraciones) de objetos o cosas que puedan ser referenciadas una o muchas veces por los ficheros principales de escena. La mecánica de uso de los archivos de inclusión es la siguiente: en un lugar dado del archivo de escena, generalmente al principio del mismo, se suelen colocar una o más instrucciones como la que sigue: #include "tanque.inc".

Cuando ordenamos a POV generar una imagen, el primer paso que realiza es comenzar a leer el fichero escénico suministrado como entrada. Mientras el fichero se lee, POV va interpretando lo leido y creando un modelo interno de la escena. A este proceso se le llama "parsing", y "parser" a la parte de POV encargada de dicho proceso. Pues bien, cuando durante el parsing del fichero dado como entrada, el parser halla una línea como la anterior, POV carga el fichero indicado en la sentencia #include y procede a interpretarlo. Al acabar con él, el parser continúa leyendo e interpretando las líneas del fichero .pov inicial.

La sentencia #include se emplea para dar claridad a nuestros proyectos, ya que elimina la necesidad de tenerlo todo en un único fichero. Así, podríamos crear 5 tanques diferentes en otros tantos ficheros include y dejar en el fichero principal (el dado como entrada a POV) las sentencias de cámara, luces y colocación de los tanques.

Para funcionar, #include precisa que los ficheros .inc se encuentren en el directorio en curso, o bien en el direccionado por la orden de la línea de comandos +L. Povray adjunta en el directorio include varios archivos de inclusión con declaraciones de colores (colors.inc), texturas (woods, stones...), etc., que nos serán muy útiles. Sin embargo, como los elementos declarados ocupan espacio en memoria, no conviene "incluir" ficheros que no vayamos a emplear.

#Declare

Los ficheros de inclusión utilizan mucho la instrucción #declare. Esto es así porque esta sentencia ayuda considerablemente a estructurar las archivos escénicos, y hacerlos mucho más cortos y sencillos. Con #declare se puede hacer que un identificador equivalga a una larga y compleja serie de sentencias que podrán ser referenciadas más tarde cuantas veces lo deseemos, simplemente incluyendo el identificador. #Declare puede usarse para declarar un color, una textura compleja, un objeto o parte del mismo, un vector, un valor en flotante, una operación CSG, una cámara, etc. Con #declare bastará con cambiar la declaración que sigue a esta sentencia para que la alteración se haga efectiva en todos los sitios donde se haya colocado el identificador, con el consiguiente ahorro de tiempo y trabajo.

Un detalle adicional muy importante es el hecho de que la declaración de un nuevo identificador puede apoyarse en otros identificadores ya creados.

Veamos un ejemplo válido extraido de colors.inc:

```
#declare White = rgb 1
#declare Gray05 = White*0.05
```

Aquí el color blanco definido es empleado en la definición del gris. Más tarde,

Figura 1: "Palabras reservadas de Povray"

aa_level fog_offset reciprocal aa_threshold fog_type recursion_limit abs frequency red acos gif reflection acosh global_settings refraction adaptive glowing render adc_bailout gradient repeat agate granite rgb agate_turb gray_threshold rgbf all green rgbft alpha halo rgbt ambient height_field right ambient_light hexagon ripples angle hf_gray_16 rotate aperture hierarchy roughness arc_angle hollow samples area_light hypercomplex scale asc if scallop_wave asin ifdef scattering asinh iff seed assumed_gamma image_map shadowless atan incidence sin atan2 include sine_wave atanh int sinh atmosphere interpolate sky atmospheric_attenuation intersection sky_sphere attenuating inverse slice average ior slope_map background irid smooth bicubic_patch irid_wavelength smooth_triangle black_hole jitter sor blob julia_fractal specular blue lambda sphere blur_samples lathe spherical_mapping bounded_by leopard spiral box light_source spiral1 box_mapping linear spiral2 bozo linear_spline spotlight break linear_sweep spotted brick location sqr brick_size log sqrt brightness looks_like statistics brilliance look_at str bumps low_error_factor strcmp bumpy1 mandel strength bumpy2 map_type strlen bumpy3 marble strlwr bump_map material_map strupr bump_size matrix sturm camera max substr case max_intersections superellipsoid caustics max_iteration switch ceil max_trace_level sys checker max_value t chr merge tan clipped_by mesh tanh clock metallic test_camera_1 color min test_camera_2 color_map minimum_reuse test_camera_3 colour mod test_camera_4 colour_map mortar text component nearest_count texture composite no texture_map concat normal tga cone normal_map thickness confidence no_shadow threshold conic_sweep number_of_waves tightness constant object tile2 control0 octaves tiles control1 off torus cos offset track cosh omega transform count omnimax translate crackle on transmit crand once triangle cube onion triangle_wave cubic open true cubic_spline orthographic ttf cylinder panoramic turbulence cylindrical_mapping pattern1 turb_depth debug pattern2 type declare pattern3 u default perspective ultra_wide_angle degrees pgm union dents phase up difference phong use_color diffuse phong_size use_colour direction pi use_index disc pigment u_steps distance pigment_map v distance_maximum planar_mapping val div plane variance dust png vaxis_rotate dust_type point_at vcross eccentricity poly vdot else polygon version emitting pot vlength end pow vnormalize error ppm volume_object error_bound precision volume_rendered exp prism vol_with_light exponent pwr vrotate fade_distance quadratic_spline v_steps fade_power quadric warning falloff quartic warp falloff_angle quaternion water_level false quick_color waves file_exists quick_colour while filter quilted width finish radial wood fisheye radians wrinkles flatness radiosity x flip radius y floor rainbow yes focal_point ramp_wave z fog rand fog_alt range

al estudiar town.inc, veremos como se emplea #declare para definir objetos simples usados en la construcción de otros más complejos (por ejemplo, ventanas para las paredes, grupos de paredes para los pisos, pisos para las casas y grupos de éstas para un pueblo).

Otra caracteristica muy importante de #declare es que los identificadores creados pueden ser redeclarados utilizando el valor anterior del identificador. Asi, por ejemplo, es posible escribir...

```
#declare X=5
#declare x1=X+7
#declare total=x1*(X+10)
```

Donde el valor de "total" es de 180. La utilidad de todo esto se hará evidente luego, cuando estudiemos las sentencias #if, #while y otras. Una declaración siempre empieza, como podemos ver, con la palabra #declare seguida del identificador. Luego, se pone un signo '=' al que seguirá la definición de la declaración. (Recordemos otra vez que POV distingue entre mayúsculas y minúsculas. No será igual "total" que "Total"). Los identificadores creados han usarse de manera lógica. Es decir, si el identificador es un objeto, tendrá que ser referenciado dentro de una palabra object. Si es una textura, tendrá que colocarse dentro de una sentencia texture, etc. Veamos un ejemplo:

```
#declareP_Gold2=rgb
CVect2
#declare F_MetalA =finish {
  brilliance 2
  diffuse D_GoldA
  ambient A_GoldA
  reflection R_GoldA
  metallic M
  specular 0.20
  roughness 1/20
  }
#declare T_Gold_2A =
texture { pigment { P_Gold2
} finish { F_MetalA } }
```

Como puede verse, la definición de la textura T_Gold_2A emplea el color P_Gold2 y el acabado F_MetalA (finish) definidos anteriormente. (Para la definición de estas características, la textura emplea las palabras pigment y finish, de las que hablaremos más adelante).

La declaración de un objeto tampoco tiene misterios:

```
#declare bola=sphere{
<0,0,0>, 1 }
```

El objeto declarado, ya sea sencillo como bola o muy complejo, puede ser invocado después en tantos sitios como deseemos empleando la sentencia object del siguiente modo:

```
object { bola
texture{T_Gold_2A} scale
<2,1,3>}
object { bola
texture{T_Silver_4A} transla-
te <10,20,30> }
```

Nótese como el identificador del objeto bola es usado para crear dos objetos situados en posiciones y escalas diferentes, y que pueden llevar texturas distintas. (siendo una de estas, la declarada en el ejemplo anterior).

Tipos de objetos de POV

Llamaremos figuras primitivas -o primitivas a secas- a los objetos simples que se definen con las sentencias reservadas para ello como sphere, box, plane y otras. Aquellos objetos más complejos, resultantes de las operaciones hechas entre objetos, no serán considerados como primitivas. A su vez, las primitivas pueden ser finitas o infinitas. Una esfera es un objeto finito, ya que siempre tiene límites definidos sin importar cuán grande pueda llegar a ser. La primitiva plane, en cambio, tiene las mismas propiedades que el plano matemático. O sea, carece de grosor y se extiende infinitamente por el espacio virtual.

La filosofía del modelado en POV se basa en operar con estas primitivas para construir objetos. Los objetos resultantes pueden sumarse o combinarse entre sí para producir objetos aún más complejos que pueden emplearse para crear modelos de mayor complejidad.

Primitivas

La esfera es una de las primitivas de más rápida interpretación de POV.

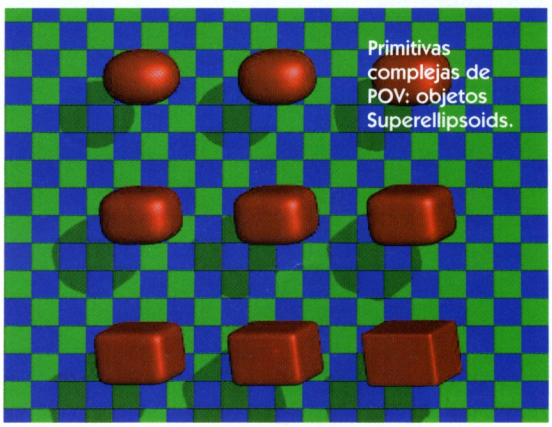
Este objeto es generado con la sentencia sphere, cuya sintaxis es: sphere{ <centro>, radio }.

<Centro> es un vector que define los valores de X,Y y Z para la localización espacial del centro de la esfera.

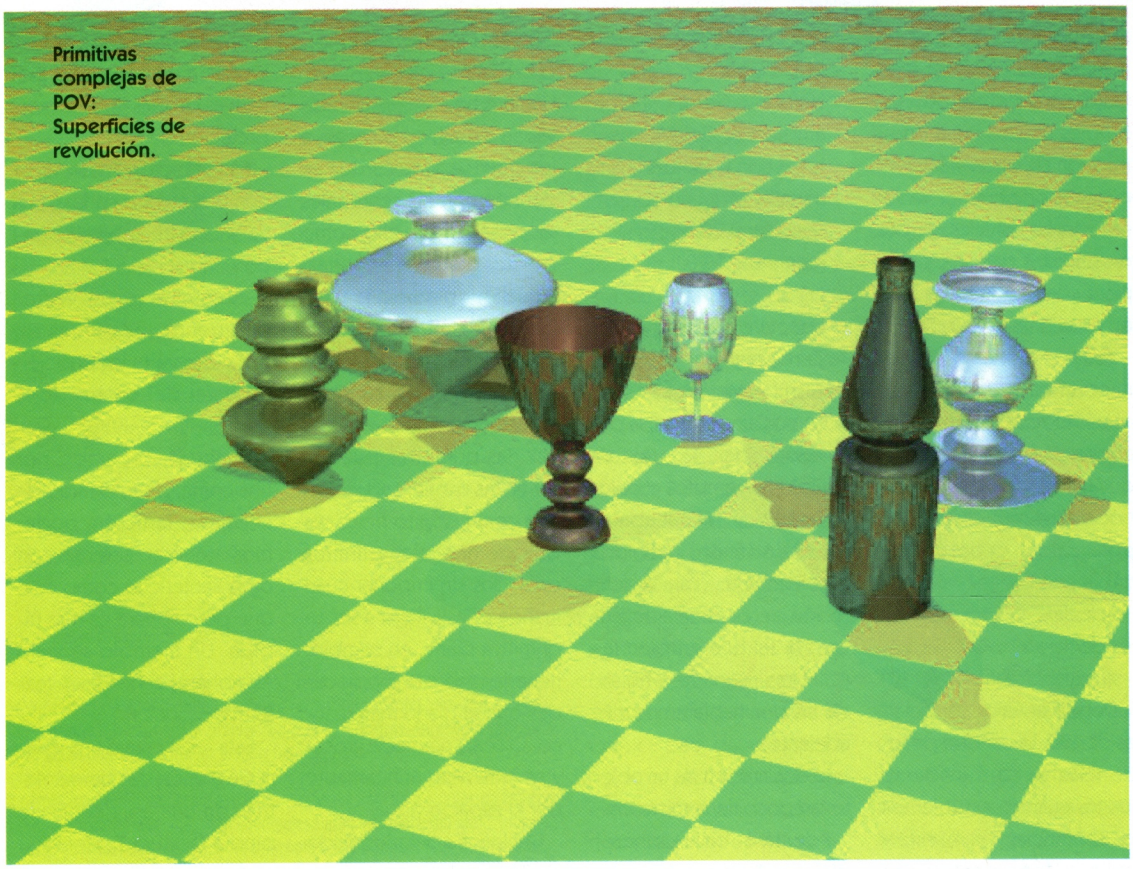
Después, separado por una coma, sigue el radio de la esfera. También, podemos generar esferas con otras sentencias como cuadric pero sphere es más rápida. Un ejemplo sencillo es: sphere{<-12,4,0>,4 texture{T_Stone23} }.

Esta línea creará una esfera de 8 unidades de diámetro con una apariencia de piedra (textura T_Stone23) en la posición indicada por el vector.

Otra primitiva sencilla y de rápido dibujo es box (caja). Un ejemplo sencillo de box sería un cubo o, dado que la longitud de la forma no tiene porque ser la misma en los tres ejes,



Primitivas complejas de POV: objetos Superellipsoids.



Primitivas complejas de POV: Superficies de revolución.

una caja de zapatos. La figura se define especificando las posiciones espaciales de dos vértices opuestos de la misma. Ambos vértices son definidos por dos vectores separados por comas, con lo que la sintáxis de la caja queda como sigue: box{<esquina1>,<esquina2> }.

También se supone que cada uno de los lados es siempre paralelo a uno de los ejes de coordenadas. Sólo posteriormente, con el empleo de sentencias de rotación, esta situación inicial puede alterarse. Otro detalle importante a tener en cuenta es que el primer vector de la definición debe referirse siempre al punto con los valores X,Y y Z más pequeños.

Un ejemplo: box{<-25,0,0> , <25,50,5> texture{textmur} }.

La siguiente primitiva a estudiar es el cilindro. La sentencia empleada para crear estos objetos es cylinder, y su sintaxis es...

cylinder{<base>, <cima>, radio }

Los dos vectores indican las coordenadas X,Y y Z del punto central de cada extremo del cilindro (de la base y la cima).

El valor de "radio" especifica el radio del objeto. El cilindro creado aparecerá como un objeto "solido" a menos que escribamos la palabra "open" después del radio. En este caso, se eliminarán los planos que forman las dos bases del cilindro, con lo que éste quedará hueco y sus paredes carecerán de grosor. Debido a este último detalle no se empleó open en los cilindros que definían las torres de castlib, ya que se pretendía que las torres, aunque huecas, tuviesen muros de un grosor dado. Por esta razón, la forma básica de las torres se creó usando operaciones CSG entre cilindros.

(Ver PCmanía número 47). Aquí tenemos un ejemplo de cylinder:

cylinder {<6,4,0>, <6,14,0>, 4 pigment {Red} }

(Pigment se emplea para crear una textura consistente en un color para el cilindro).

Continuemos con el siguiente objeto: el cono. Su filosofía y sintáxis son muy similares a las del cilindro. La única diferencia estriba en que ahora cada uno de los extremos del objeto lleva un radio propio. También podemos emplear la palabra open después del segundo radio con el mismo resultado que antes. La sintáxis para este objeto es...

cone {<base>, radio_base, <techo>, radio_techo }

...y el ejemplo correspondiente...

```
cone{
  <-12,1,0>,5 // Centro y radio de uno de los extremos
      <-12,8,0>,3 // Center y radio del otro extremo
      pigment {Cyan}
  }
```

Si se desea que el cono acabe en un único punto, debe dársele radio cero a uno de los extremos.

Objetos infinitos, superficies matemáticas y plane

Todos los objetos estudiados hasta ahora son finitos pero POV permite crear objetos con superficies infinitas. Tal es el caso del plano, el cual es sumamente útil en las operaciones CSG. Otras primitivas que pueden caer en esta categoría son cubic, poly y quartic, las cuales describen objetos basados en ecuaciones polinómicas. El cálculo de estas superficies es lento y a veces no muy fiable. Además, predecir la forma final exacta de este tipo de objetos es difícil, por lo que estas primitivas sólo suelen ser usadas por programas de modelado. Por estas razones nos limitaremos a estudiar la primitiva "plane".

La palabra "plane" sirve para crear un plano, infinito en dos de los ejes y carente de longitud en el eje restante. Dicho plano puede ser utilizado como suelo o pared y puede recibir texturas en su superficie, o bien ser utilizado, como veremos más adelante, en operaciones de CSG. El plano tiene dos posibles formatos, ambos igualmente válidos, el primero y más antiguo es: plane { <x,y,z>, distancia }.

Donde el vector <x,y,z> se refiere a la normal al plano de superficie.

(La normal a un plano es un vector -puede visualizarse como una flecha- que abandona el plano de manera perpendicular a éste, o sea como un poste en relación al suelo. Sabiendo esto es fácil imaginar que un vector paralelo al eje Y puede servirnos para crear un suelo mientras que otro paralelo al eje Z podría utilizarse para dibujar un muro paralelo al plano que forma la pantalla). El parámetro siguiente es un valor en flotante que indica la distancia del plano al origen de coordenadas. De esta forma el ejemplo siguiente...

```
plane { <0,1,0>,5 }
```

...indica que la normal será perpendicular al plano formado por los ejes X y Z, con lo que obtendremos un plano paralelo a dichos ejes y situado a 5 unidades por encima del origen de coordenadas.

El signo del valor correspondiente al eje sirve tanto para indicar el sentido en que la normal se aleja del origen como para saber en qué lado del eje deben contarse las unidades para situar el plano.

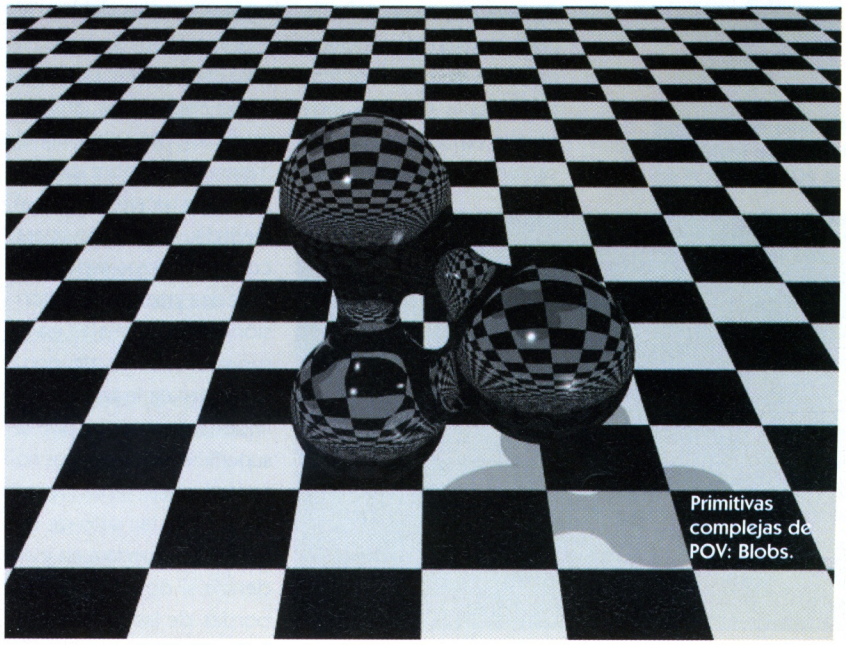
Dado que también podemos indicar valores negativos en el valor que sigue al vector, la decisión acerca del lado del eje en que finalmente debe situarse el plano, puede parecer un tanto liosa.

Afortunadamente, bastará con que recordemos que el lado, positivo o negativo, se obtiene sumando los signos del eje y el valor de distancia. O sea: positivo y positivo dará positivo. Al igual que negativo más negativo.

Otra combinación nos dará el lado negativo del eje.

En la nueva sintáxis, más concisa, el ejemplo equivalente sería: plane{y,5}. Donde el vector ha sido sustituido por el nombre del eje paralelo a la normal, lo que resulta mucho más intuitivo.

Posteriormente, en las operaciones CSG, volveremos sobre plane y comprobaremos un detalle bastante curioso: los planos (al menos en POV), tienen un "interior" y un "exterior".



Primitivas complejas de POV: Blobs.

Texturas, colores y

Ya sabemos bastante acerca de cómo crear objetos. De hecho, aunque no lo parezca, todos los modelos de Castlib se han creado partiendo de operaciones entre las primitivas ya estudiadas (también se ha empleado prism para puertas y ventanas).

Aún no se ha tocado el tema de la apariencia de los objetos a pesar de que este asunto es de una importancia capital, ya que sin una textura que describa el material de una superficie, no sera visible.

Materiales

Comencemos con ello pues: para crear la apariencia de un material, el potente lenguaje de POV nos ofrece varias docenas de sentencias que emplearemos para definir los elementos constituyentes de las texturas: pigmentos, normales, acabados y halos. Un pigmento es el color o patrón de colores del material o bien el bitmap que lo recubre. El pigmento se define con sentencias englobadas dentro de las llaves de la sentencia pigment. Hay que decir que estos colores son los "reales" del objeto pero, bajo ciertas condiciones, no siempre serán los que se aprecien al obtener la imagen. Por ejemplo, si un objeto tiene un pigmento blanco y colocamos una luz verde cerca de él, el objeto presentará un aspecto verdoso. En cuanto a "normal", la sentencias escritas dentro de esta instrucción se emplean para crear apariencias tales como abolladuras, anillos, ondulaciones, etc., en el material. El nombre de esta sentencia hace referencia al método empleado para conseguir estos aspectos, logrados gracias a la distorsión de las normales en la superficie del objeto donde se aplica la textura. Es importante recordar que la superficie de los objetos no se vuelve realmente irregular. Se trata tan sólo de un truco que quedará en evidencia si observamos los bordes de las formas que empleen texturas de este tipo. El siguiente apartado es el acabado, el cual se obtiene gracias a las sentencias empleadas para la instrucción finish. (Aquí la palabra acabado alude a las propiedades reflectivas y refractivas del material). Por último, el apartado halo, incorporado a partir de la versión 3.0 de POV, se emplea para simular efectos de nubes, niebla, fuego, etc., dentro de los objetos.

Una pov-textura puede ser de tipo sencillo o de tipo especial. A pesar de su nombre, las texturas de tipo sencillo pueden ser muy difíciles de crear, dependiendo de lo que pretendamos hacer. El formato completo para una textura sencilla es:

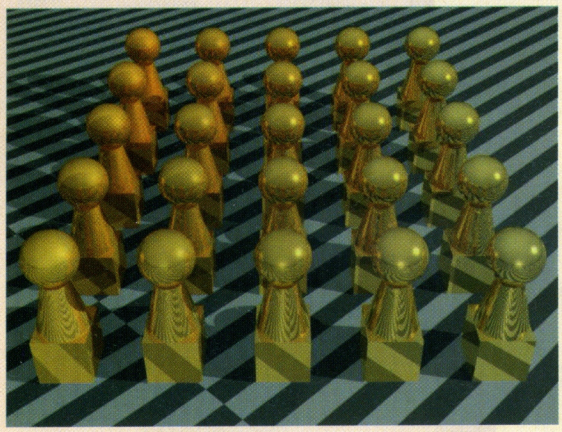
```
texture {
  identificador_de_Textura
    pigment {..}
    normal {..}
    finish {..}
    halo {..}
    Transformaciones
  }
```

Las líneas anteriores podrían incluirse dentro de las llaves de un objeto de este modo:

```
box{<-4,2,-4>,<4,10,4>
        texture{definición_de_textura}
    }
```

Sin embargo, es mucho más cómodo declarar un identificador para la textura.

Ejemplo de la librería de texturas de POV:oro.



transformaciones

De este modo, la textura definida podrá ser empleada por tantos objetos como queramos sin necesidad de escribir todas las veces la misma definición. Por ejemplo, podríamos escribir:

```
#declare madera=texture{definición_de_textura}
box{<-4,2,-4>,<4,10,4>
    texture{madera}
    }
sphere{<12,15,23>, 4
texture{madera} }
```

Por otro lado, cada una de las líneas que anteriormente vimos en el formato completo de la declaración de la textura pertenece a un apartado opcional en dicha declaración, aunque hay una excepción para esto: si "identificador_de_Textura" no se incluye, deberemos forzosamente dar un pigmento a la textura. De hecho, existe una forma abreviada de definir una textura donde sólo se precisa definir el pigmento. Así, la siguiente línea es correcta:

```
sphere{<2,75,13>, 6 pigment{White} }
```

Realmente, lo que sucede es que se aplican los valores de la textura por defecto para normal y finish. En caso de que pretendamos crear nuestra propia textura por defecto, podemos hacerlo usando la sentencia #default.

La siguiente línea, por ejemplo, cambiaría el valor por defecto de finish:

```
#default{finish{ambient .7} }.
```

Antes de continuar, conviene tomar nota de una sentencia usada en pigment que nos será muy útil en las pruebas. Se trata de checker. Esta instrucción crea una trama de pigmentación tridimensional compuesta por cubos de colores alternos. La longitud de las aristas de cada cubo de color es de una unidad y la orientación de los cubos es paralela a los ejes del universo virtual.

La importancia de checker radica en que equivale a una especie de grid tridimensional: un fondo con el que nos será fácil comprobar si son correctas la colocación, orientación y escala de los objetos en construcción. El autor de estas líneas emplea mucho esta sentencia en la aplicación de las primitivas con las que hace el suelo sobre el que irán los objetos en fase de desarrollo. Más tarde veremos muchos ejemplos de esto.

Por ahora, veamos un ejemplo de checker:

```
box{<-4,-4,-4>,<4,4,4>
    pigment{checker pigment{Red}, pigment{Yellow}}
    }
```

Esto puede abreviarse:

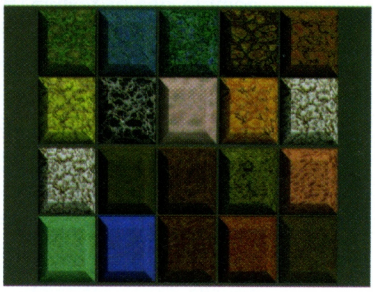
```
box{<-4,-4,-4>,<4,4,4>
    pigment{checker Red,
Yellow}}
    }
```

Al igual que otras sentencias usadas en texturas, checker no limita su uso a un sólo tipo de componente de texturas. También puede ser empleada para alterar las normales del material. Por último, los pov-expertos deben tomar nota de que checker ha reemplazado a tile para crear patrones ajedrezados de texturas. Por ejemplo...

```
plane{y,2 texture { checker texture{T_Wood3}, texture{T_Wood2}} }
```

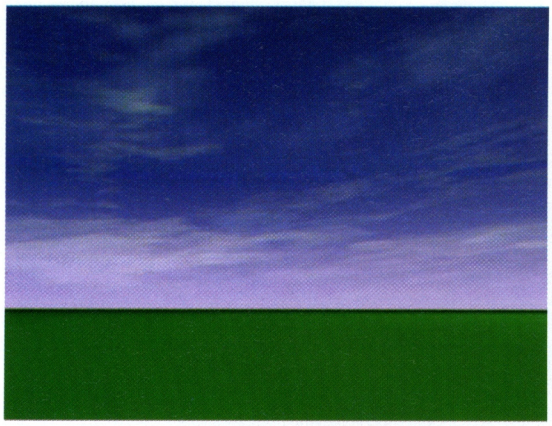
...crea un tablero de ajedrez de infinitas casillas alternando el uso de dos texturas de madera de la librería.

Otro detalle muy importante para la declaración de texturas es el hecho de que podemos aprovechar texturas ya creadas para definir las nuevas (lo cual también es válido para el caso de los pigmentos). En el formato completo de la decla-



Ejemplo de la librería de texturas de POV: piedra y madera.

Ejemplo de la librería de texturas de POV: cielo.

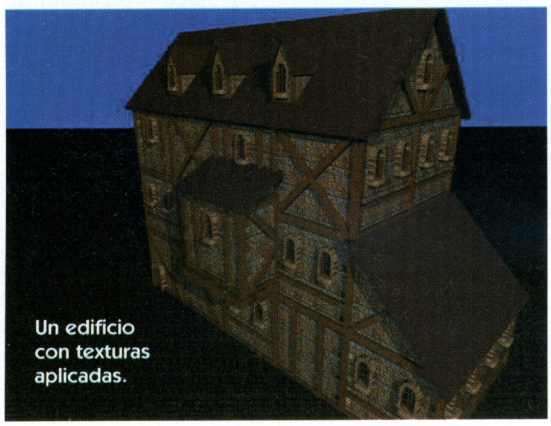


ración de textura sencilla que estudiamos anteriormente figuraba la palabra identificador_de_Textura, la cual es, sencillamente, el nombre de una textura ya declarada antes. Al incluir el identificador de una textura ya creada al principio de nuestra declaración, automáticamente nuestra textura pasará a tener los mismos componentes pigment, normal, finish y halo de aquella. Cualquier cambio que especifiquemos para dichos componentes en nuestra declaración se "sobreimprimirá" en los valores aportados por el identificador de textura incluido. Como podéis imaginar, esto es de una importancia capital, ya que con POV se adjunta una completísima librería de texturas de rocas, metales, maderas, cielos, cristales... En las siguientes líneas, por ejemplo, se declara una nueva textura de madera que es igual a la predefinida T_Wood17, pero cambiando el valor de turbulencia:

```
#declare T17 = texture
{ T_Wood17
  pigment{
      wood
      turbulence 0.0575
      }
}
```

Podemos pues limitarnos a utilizar "tal cual" las texturas de la librería (simplemente empleando la sentencia #include con los archivos correspondientes) o bien podemos incluirlas para usarlas en la declaración de nuestros propios materiales o bien podemos crear de cero nuestras texturas.

En cuanto a las texturas "especiales", estas son combinaciones de dos o más texturas simples con las que puede crearse una trama o patrón. La explicación detallada de las sentencias de texturas y el estudio de su aplicación para crear materiales complejos rebasa el propósito de este libro. Por tanto recomiendo a los pov-novatos que, al menos de momento, se limiten a emplear las texturas predefinidas o bitmaps propios. No obstante lo que ya hemos visto aquí sera sufiente para las escenas de prueba.



Un edificio con texturas aplicadas.

Transformaciones espaciales

Otro aspecto básico en el diseño de los objetos es el uso de transformaciones espaciales. Llamamos así a las operaciones que pueden efectuarse sobre los objetos para alterar su posición espacial, sus dimensiones o su orientación. POV dispone de tres tipos de transformaciones: la traslación, la escalación y la rotación. La primera operación cambia la colocación de los objetos, la segunda altera las dimensiones de los objetos y la tercera sirve para girar la orientación de los objetos. Los tres tipos de transformaciones necesitan de un vector que especifique los valores necesarios para los ejes X, Y y Z. Con ello, el formato de estas operaciones es:

```
rotate <x,y,z>
scale <x,y,z>
translate <x,y,z>
```

Podemos utilizar tantas operaciones de transformación como queramos para un mismo objeto y emplear, asimismo, el orden que queramos para ellas. Por ejemplo, es perfectamente válido escribir:

```
sphere { <0, 0, 0>, 3
    texture { T17 }
    scale <1,2,1>
    rotate <0,0,45>
    translate <-5, 20, 1>
    rotate <35,0,0>
}
```

Como veremos, las transformaciones pueden aplicarse al objeto completo, o sea: a su forma y textura o bien podemos hacer que afecten sólo a la geometría o a la textura del objeto. En el ejemplo anterior, las operaciones afectan a ambas cosas.

Traslación

Ésta es la más simple de las transformaciones. Los valores en cada eje, en el vector, especifican el número de unidades en que el objeto se desplazará a lo largo de los ejes de coordenadas. Estos valores no son absolutos, sino relativos a la última posición del objeto. ¡Recordar este detalle es vital! Por ejemplo, en el ejemplo siguiente, el centro de la esfera no acaba en la posición <2,10,4> sino en la <3,15,-2>.

```
sphere{<5,5,5>,2 pigment
{Red} translate<-2,10,-7> }
```

Escalación

Con scale podemos alterar las dimensiones de los objetos usando factores de escalado como parámetros para los ejes. Si los factores tienen el mismo valor en los tres ejes, las dimensiones del objeto afectado aumentarán o disminuirán pero la forma del objeto quedará igual. El factor de escala multiplica las dimensiones del objeto a lo largo del eje correspondiente. Así...

```
sphere{<0,0,0>,2 pigment {Red} scale<.5,2,1> }
```

...la esfera del ejemplo se estrechara en el eje X, quedando con una longitud de 2 unidades. En el eje Y, en cambio, la esfera se alargará hasta las 8 unidades de longitud y la longitud en el eje Z quedará igual, ya que hemos multiplicado su valor

original por 1 unidad. Es muy importante tener en cuenta que los cambios de escala son relativos al último tamaño obtenido (se puede haber empleado más de una sentencia scale) y también a la posición del objeto. Por ejemplo, en el siguiente caso...

sphere{<0,5,0>,2 pigment {Red} scale<.5,2,1> }

...comenzamos con una esfera centrada en X y Z, cuyo centro en el eje Y es la posición 5. Como la forma de la esfera se multiplica por 2 en este último eje, el nuevo centro se desplazará a la posición 10, con lo que habrá un desplazamiento del objeto en este eje. En los ejes restantes, la esfera no sufrirá cambios con respecto al ejemplo anterior, ya que el objeto estaba centrado en dichos ejes. Por esta razón, es practicamente obligatorio efectuar la "construcción" de objetos complicados fijando su punto central (de los tres ejes) en la posición <0,0,0>. Por ello, si lo que queríamos era que la esfera del primer ejemplo quedara centrada en <0,5,0>, deberemos escribir:

sphere{<0,0,0>,2 pigment {Red} scale<.5,2,1> translate<0,5,0>}

Rotación

La instrucción rotate girará el objeto el número de grados que se indique en los parámetros para X, Y y Z. El giro se efectúa siempre con respecto al centro de coordenadas. Esto quiere decir que si el objeto está centrado en dicho centro, rotará sobre sí mismo. En cambio si se halla a cierta distancia del centro parecerá describir una órbita en torno a la posición <0,0,0>. Por esta razón, si por ejemplo queremos que un objeto centrado en la posición <21,-56,17> gire 45 grados en el eje X sin alterar su posición, deberemos escribir...

translate<-21,56,-17>
rotate<45,0,0>
translate<21,-56,17>

La primera traslación posiciona el objeto en el centro de coordenadas, la siguiente rota el objeto los grados deseados y la tercera lo devuelve a su posición original (pero ya rotado). Nótese que realmente no es preciso centrar el objeto para el eje Y.

Los valores dados a rotate pueden ser positivos o negativos, lo que afectará al sentido del giro. Para recordar el sentido que tendrá una rotación positiva para cada eje, existe la siguiente regla. Mentalmente haremos el gesto de OK con la mano izquierda. Seguidamente, sin alterar el gesto, orientaremos la mano de forma que el pulgar apunte en el sentido positivo del eje cuyo sentido de giro intentamos recordar. El sentido positivo del giro será el mismo que siguen nuestros 4 dedos cerrados (y el negativo el opuesto, por supuesto). Por esta razón el sistema de coordenadas de POV es llamado "de mano izquierda". Ya sólo queda advertir que el orden de las rotaciones influirá en el resultado final. En la instrucción: rotate<45,10,20>.

El objeto rotaría primero 45 grados en X, luego 10 en el Y y finalmente 20 en el Z. Si deseamos que el orden de las rotaciones sea distinto habremos de usar dos o tres sentencias de rotación.

Operaciones CSG entre objetos

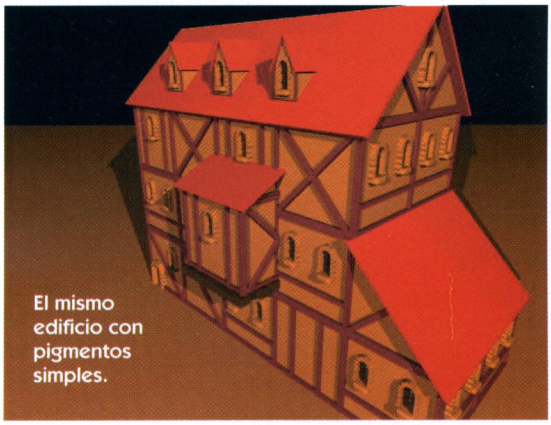
Nuestras escenas no podrían ser demasiado vistosas si se limitaran a mostrar objetos como cubos y esferas con distintos tamaños, colores y posiciones.

Afortunadamente, POV cuenta con un factor adicional que es la pieza angular en la que reposa la potencia del diseño de objetos: las operaciones CSG.

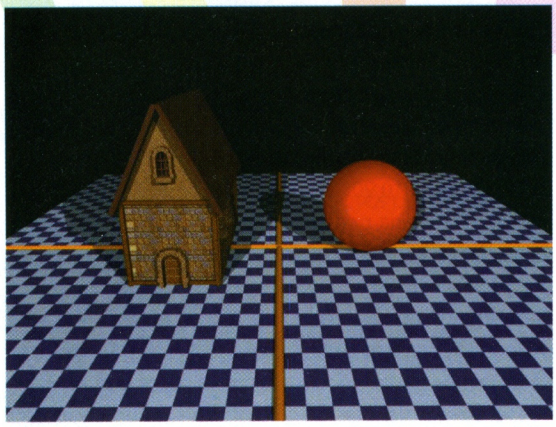
Estas operaciones, también llamadas booleanas, se emplean para crear nuevos objetos realizando sumas y restas con los volúmenes espaciales de dos o más objetos. Supongamos que tenemos dos objetos; una esfera y un cubo colocados de tal manera que el cubo (de mayor tamaño) se superpone al volumen espacial ocupado por la mitad inferior de la esfera (comparten dicho espacio). Si restamos el cubo a la esfera, el resultado será una semiesfera. Si la resta se efectúa sobre el cubo el resultado será un cubo con un espacio esférico (recortado en su interior).

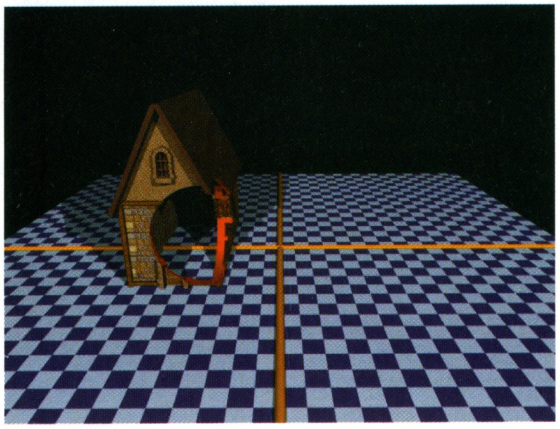
Las transformaciones espaciales estudiadas antes pueden aplicarse también a los objetos CSG. Pero lo más importante es que estos objetos pueden, a su vez, ser usados en nuevas operaciones de CSG para producir nuevos objetos.

De esto se deduce que, partiendo de primitivas muy sencillas (cajas, esferas, cilindros, etc.), podemos acabar obteniendo modelos de apariencia muy compleja. Este modo de trabajo también es llamado "Geometría de construcción de sólidos" porque sólo puede operar con ob-



El mismo edificio con pigmentos simples.





La esfera roja alrededor del eje Y hasta superponerse a la casa, sobre la que efectuará una resta CSG.

jetos que tengan un volumen interior definido. Las cajas y las esferas, por ejemplo, lo tienen, al igual que las primitivas de cilindros y conos -siempre que estas no empleen la palabra "open" (ya que entonces los objetos estarían abiertos y sin grosor en las paredes).

POV dispone de 4 operaciones de CSG: union, difference, intersection y merge.

Unión CSG

Con esta operación, dos o más objetos se unen para formar uno sólo. Para crear una unión bastará con englobar dos o más objetos dentro de las llaves de la palabra unión.

Por ejemplo:

```
union{
        box{<-5,-4,-10>,
<5,4,10> }
        sphere{<0,0,0>,4
translate<10,15,0> }
    pigment{Red}
    rotate <30,0,0>
}
```

En esta unión, la operación de traslación sólo afecta a la esfera pero la rotación afecta a ambas formas y el pigmento se usa también para las dos.

También podemos crear un identificador para esta unión y emplearla en más de un sitio (otra vez declare). La mayoría de las veces la unión se emplea para crear un único objeto sobre el que se operan nuevas operaciones CSG.

Difference CSG

La palabra difference equivale a la resta de objetos. En esta operación la resta se efectuará sobre el primer objeto citado y los siguientes se emplearán como objetos de resta. Veamos un ejemplo:

```
difference{
    sphere{<0,0,0>,4 }
     box{<-10,1.50,-10>,
<10,10,10> }
     box{<-10,1.50,-10>,
<10,10,10>          rotate
<180,0,0> }
    rotate <0,0,90>
    pigment{Red}
}
```

Aquí el resultado será una especie de llanta: la esfera de la que se restan las dos cajas. Nótese como ni tan siquiera nos molestamos en calcular los extremos de la caja inferior.

Simplemente creamos la misma caja y la hacemos rotar 180 grados en X para que quede a igual distancia en el lado negativo del eje Y.

Finalmente, la segunda rotación afecta al objeto creado y lo gira para que la llanta pueda "rodar".

Intersection CSG

En la intersección, el objeto resultante será el formado por los volúmenes espaciales que compartan todos los objetos usados en esta operación. Las siguientes líneas:

```
    intersection {
      sphere { <0, 0, 0>, 4
translate <-2,0,0> }
      sphere { <0, 0, 0>, 4
translate <2,0,0> }
     pigment { Red }
    }
```

...crean una especie de balón de rugby creado con el volumen espacial que ambas esferas compartían.

Merge CSG

Esta operación funciona exactamente igual que unión.

La única diferencia estriba en que las partes compartidas de los objetos usados con merge (fusion) desaparecen.

Esto es útil cuando se estan aplicando texturas no opacas (cristal, hielo, etc.), donde puedan verse partes internas.

Planificar un proyecto

Al idear un proyecto para crear escenas o animaciones es preciso tener en cuenta una serie de factores que influirán en el buen desarrollo del mismo. En primer lugar, hay que tener en cuenta la escena en sí; qué objetos van a componerla, cómo vamos a construirlos, las texturas que se emplearán...

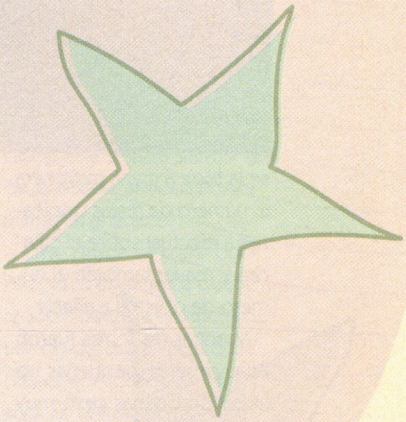
A partir de esto, hay que intentar adivinar si la realización del proyecto es factible, teniendo en cuenta el hardware y el tiempo de que se dispone. Otro factor adicional que puede acabar siendo un verdadero problema para algunas personas es el gusto por el detalle. Uno puede comenzar con una idea relativamente sencilla como la de un castillo compuesto por 2 o 3 torres sencillas, un par de muros y un montecillo donde colocarlo todo. Sin embargo, conforme se van creando objetos, uno puede acabar entusiasmándose demasiado, y acabar intentando crear una auténtica ciudadela amurallada, con docenas de casas distintas y un enorme castillo de fondo.

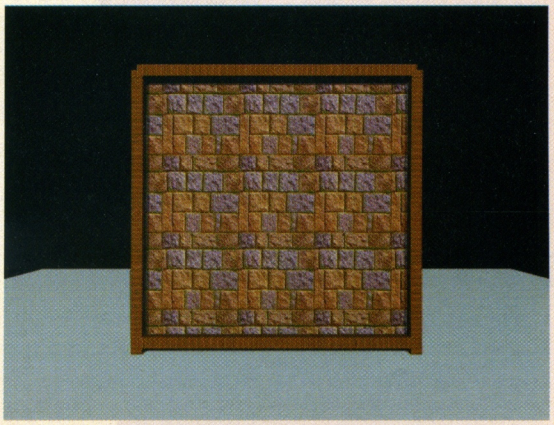
Esto fue lo que sucedió en el caso de Castlib. En un momento dado, pensamos colocar casas alrededor del castillo para darle mayor realismo. Estas casas tan sólo servirían de fondo para la estructura principal (el castillo) y nunca se verían de cerca, por lo que no precisarían demasiado detalle. El caso es que, sin saber como, al final las casas comenzaron a recargarse con detallitos, variaciones, texturas propias, etc. El tiempo se acababa y las estructuras del castillo... ¡Por el rayo trazado con calidad 9! No estaban listas. Pero, en fin, de todos modos town.inc (fichero extraido del proyecto original) nos servirá para ilustrar los pormenores de la creación de un proyecto y algunas nuevas características y sentencias de POV 3.0. Antes de comenzar, no obstante, queremos dejar claro que, en parte, lo que se dice en este capítulo es bastante subjetivo. Vamos a estudiar el desarrollo de un proyecto, pero siempre hay más de una manera de hacer las cosas y otros usuarios de POV pueden emplear métodos distintos y no necesariamente inferiores.

Evolución de un proyecto

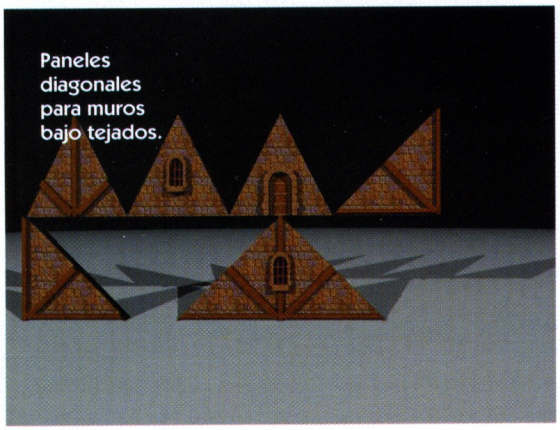
A pesar de su antiguedad, la idea inicial de Castlib no ha cambiado gran cosa. Dicha idea consistía en diseñar unas cuantas estructuras sencillas que se combinarían entre sí de diversos modos para crear otras algo más complejas, las cuales, a su vez, también podrían combinarse para crear construcciones de complejidad aún mayor. De este modo, se esperaba que las escenas aparentasen una complejidad algo elevada a cambio de un esfuerzo de modelado relativamente bajo.

Lo que sí ha experimentado cambios en Castlib ha sido la forma de construir los objetos básicos. Al principio, por ejemplo, las torres circulares se construían utilizando un enorme cilindro como cuerpo. Este cilindro podía tener 10, 20, 30 o incluso más metros virtuales. A este objeto se le practicaban restas CSG (differences) para las ventanas, se le añadían (union) marcos para estas, aspilleras para las almenas, etc. Naturalmente, cuanto mayor fue-





Este es el panel básico con el que se construyen los demás.



se la torre a crear, mayor era el número de restas booleanas a efectuar sobre el cilindro y mayor también el número de objetos a añadir.

Mientras las torres fueron relativamente pequeñas no hubo problemas pero muy pronto quedó patente una cosa: con torres relativamente grandes, llenas de ventanas y torrecitas adosadas, el tiempo de generación de la escena se disparaba. El proyecto no resultaba práctico. Por aquel entonces (usando la versión 2.0) no empleábamos bounding_slab. Al pasar a utilizar dichas figuras, los tiempos de generación disminuyeron bastante, pero pronto, cuando las escenas aumentaron un poco el nivel de detalle, se comprobó que esto era insuficiente. El problema estaba relacionado con un sistema empleado por POV para acelerar los tiempos de cálculo.

Bounding_slab

Una de las cosas para las que POV requiere más tiempo de cálculo es la comprobación -para cada rayo a trazar- de si hay o no intersección con las superficies de los objetos. Hay que pensar que por cada pixel se lanza un rayo que POV debe comprobar con todos los objetos de la escena. Además, se realizarán comprobaciones adicionales si hay que estudiar reflexiones, refracciones, etc.

Como siempre, la mejor manera de ahorrar tiempo es hacer los menos cálculos posibles y, para ello, POV emplea el truco de los objetos bounding_slab. Los boundig_slab son objetos invisibles, normalmente esferas o cajas, que POV o el usuario definen alrededor de los objetos complejos de la escena. Al usar esta técnica, POV comprueba primero si el rayo intersecciona con alguna bounding_slab y de ser así pasa a realizar los cálculos de intersección con el objeto englobado por la bounding detectada. Si no es así, el rayo se desestima. Como es mucho más rápido calcular intersecciones para una caja o una esfera que para los modelos más complejos que los objetos bounding suelen recubrir, el ahorro de tiempo se hace considerable. Otro detalle a tener en cuenta es que el bounding puede hacerse anidado. Es decir, puede haber un objeto bounding que recubra un modelo complejo y, a su vez, otros bounding interiores que recubrirán partes del modelo.

Naturalmente, para que esta técnica resulte efectiva, es preciso que los objetos bounding sean objetos rápidos de calcular (cajas o esferas) y que se ajusten lo mejor posible a los objetos que engloban. Si el recubrimiento no es perfecto (o sea: si lo hacemos manualmente y calculamos mal las dimensiones o colocación del objeto bounding), podemos perder tiempo de cálculo o peor aún; hacer que queden invisibles partes de la escena.

POV realiza un bounding automático que se ha ido haciendo más y más efectivo en cada nueva versión, pero que aún no es perfecto, ya que el bounding automatico de POV no funciona bien en ciertos casos, sobre todo en objetos CSG con los que se han empleado objetos infinitos.

Si el usuario desea colocar manualmente los objetos bounding, puede usar la sentencia bounded_by, la cual tiene el siguiente formato:

bounded_by { object_ bounding { ... } }

Y en el manual de POV se cita el siguiente ejemplo...

```
intersection {
sphere { <0,0,0>, 2 }
plane { <0,1,0>, 0 }
plane { <1,0,0>, 0 }
bounded_by { sphere {
<0,0,0>, 2 } } }
```

Aquí el usuario ha definido una esfera situada en el origen de coordenadas y con radio 2 para englobar este objeto CSG.

Sin embargo, parece claro que, sobre todo a partir de la versión 3.0, el uso de esta sentencia será cada vez más raro. Ello se debe en parte a que el bounding automático de POV es, a menudo, más preciso y mejor que el que puede especificar el usuario y además al hecho de que pocas veces se emplean objetos infinitos en las operaciones CSG (salvo plane).

Un truco sencillo para ganar tiempo

Ahora que ya sabemos algo sobre una de las técnicas empleadas por POV para ganar tiempo volvamos a nuestro proyecto. Habíamos dicho que en las escenas con torres grandes el tiempo de cálculo se disparaba. Ello se debía, como ahora quedará claro, a que se estaba empleando un método erróneo. Cuando debamos crear estructuras que empleen un buen número de operaciones CSG no debemos crear un único objeto, sino un objeto compuesto por otros.

¿Qué quiere decir esto? En el ejemplo comentado antes, una torre era un único objeto sobre el que se practicaban restas para las ventanas, puertas y portillas, y adiciones para los marcos correspondientes. El resultado podía presentar un aspecto bastante complejo pero no por ello

dejaba de ser un único objeto. Por esta razón, dentro del volumen espacial de la torre, POV debía emplear muchos testeos para saber si el rayo estaba tocando alguno de los muchos marcos de ventana o los agujeros de estas, etc. Así, en el siguiente paso, procedimos a construir pisos que después se apilarían uno encima de otro para montar las torres. De esta manera, al dividirse cada torre en varios pisos, POV pasó a crear un objeto bounding para cada piso, aparte del empleado para la unión de todos estos pisos, o sea: la torre. Con ello, al realizarse menos comprobaciones de intersección con los detalles de la torre (marcos, agujeros, etc), el tiempo de generación se redujo mucho.

Variaciones

El descubrimiento del pequeño detalle citado más arriba fue el factor principal que llevó a tirar al cubo la versión anterior de Castlib y volver a comenzar el trabajo casi desde cero. Los diseños iniciales también fueron a parar al cubo debido al descubrimiento colateral de que, en unas estructuras que se montan a base de pisos, resulta posible combinar las construcciones de muchas maneras distintas, si las alturas y dimensiones coinciden.

Sin embargo, a fin de aprovechar a fondo la idea de la variación, se necesitaba un número mínimo de pisos bastante diferentes entre sí. Debido a esto, y pensando en matar varios pájaros de un tiro, se decidió potenciar las sencillas casas que iban a emplearse como detalles de fondo para los castillos. Algunas estructuras "colgantes" usadas para dichas casas podían emplearse también en algunas partes del castillo y, de igual manera, otras casas podían colocarse dentro del mismo. Así comenzó el proceso descrito al principio de estos artículos, que llevó a que unos textos que se esperaba llenar con escenas de castillos se llenaran en cambio de escenas de casas.

Documentación

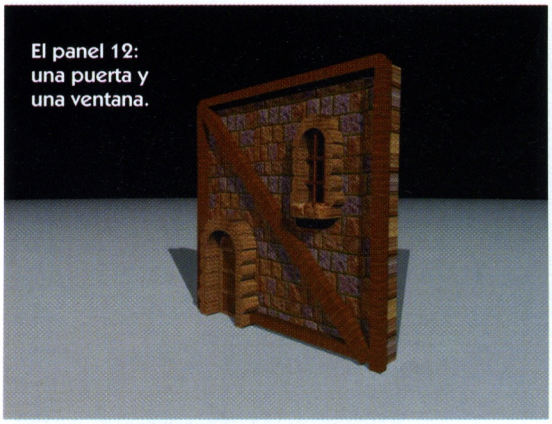
Otro aspecto fundamental a la hora de crear una escena poblada con objetos no abstractos es la documentación. Para las escenas que se deseaban crear en este proyecto no se pretendía de ningún modo ceñirse a un estilo arquitectónico dado. Lo único que se esperaba era que los modelos fuesen sencillos pero resultones y por ello la documentación se tomó de fuentes tan diversas como los espléndidos escenarios del magnífico comic "Principe valiente" de Harol Foster (una obra de arte del género), las cuidadas viñetas de "Percevan", e incluso las delirantes páginas de "Groo", de Sergio Aragonés.

También se consultó el excelente libro ilustrado; "un castillo medieval", publicado aquí por Círculo de Lectores y se examinaron otras fuentes: Portofolios de artistas como Rodney Matthews, videojuegos diversos, etc. El resultado actual podría servir quizá como fondo para un cómic informal pero no para recrear un ambiente medieval real. Para esto, sencillamente, las estructuras de castlib carecen aún del detalle suficiente. El usuario debe aceptarlas por lo que son: fondos con detalles libremente tomados y mezclados de distintas épocas y lugares.

Texturas para las casas medievales

A la hora de diseñar las casas medievales del fichero town.inc se tuvo en cuenta lo anteriormente explicado y un factor adicional; el de las texturas que iban a emplearse. POV puede crear casi cualquier tipo de textura imaginable empleando algoritmos; madera, piedra, marmol, etc. (Por ello, también se llama "procedurales" a estas texturas). El código de POV generará diversos resultados según el tino con el que hayamos empleado las sentencias de texturas que controlan a estos algoritmos. Los materiales creados de este modo con POV tienen la ventaja (sobre los bitmaps) de ser tridimensionales, es decir; no son mapas de imagen (bitmaps) que se pegan sobre los objetos. Si a una esfera a la que se le ha aplicado una pov-textura le efectuamos un "mordisco" CSG (difference), no tendremos que preocuparnos de la apariencia de la zona "mordida". Ésta seguirá siendo la de la textura aplicada originalmente al objeto.

Crear desde cero una buena textura algorítmica puede requerir bastante tiempo. Un tiempo del que a veces no se dispone. Afortunadamente, aquí se dieron dos circunstancias. La primera que POV incorpora una buena librería de texturas procedurales. Del fichero woods.inc, por ejemplo, se tomó la textura de madera empleada en diversos sitios. Otro detalle que ahorró mucho tiempo fue la obtención de dos pequeños y bonitos bitmaps de piedras de muros. Hecho esto, sólo restaba ver cómo diseñar las casas, a fin de aplicarles uno de estos bitmaps.



El panel 12: una puerta y una ventana.

Estos son los paneles muro.

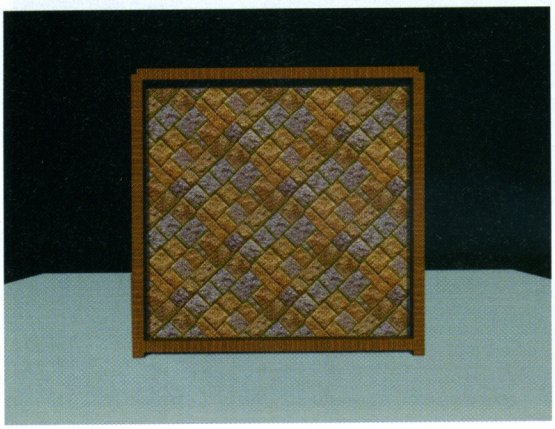
Aplicar bitmaps

La instrucción image_map se emplea para indicar a POV que los colores que se van a aplicar sobre un objeto dado corresponden a los de un fichero bitmap.

El fichero puede estar en gif, tga o en otros formatos menos conocidos y tener cualquier resolución razonable. El formato para usar esta sentencia es: pigment { image_map { formato "filename" modificadores } }

En "formato" pondremos la palabra correspondiente al tipo de archivo bitmap que vamos a usar; gif, tga, png u otros. A esto sigue, entrecomillado, el nombre del archivo, el cual será buscado por POV en el directorio en curso y en los especificados por la orden de línea de comandos L. En cuanto a los modificadores, son sentencias que afectan al modo, posición y orientación con que se aplica el bitmap sobre el objeto.

Esto es lo que sucedería con rotate zx45 dentro de la sentencia de la textura.



Dicha aplicación puede realizarse de varias maneras. Para llevarla a cabo hay que recordar que un bitmap es como una tela con la que debemos envolver, de la manera apropiada, el objeto. Por defecto, la textura se proyecta en el plano formado por los ejes X e Y, en el sentido -Z (entrando en la pantalla, según el ejemplo del principio). De no ordenar otra cosa, la textura se aplicará en el área rectangular delimitada por las coordenadas (0,0) a (1,1). Este área de aplicación puede trasladarse, rotarse o escalarse sobre el plano de aplicación (que por defecto es el formado por los ejes X e Y) utilizando las mismas sentencias usadas para operar con los objetos (o sea translate, rotate y scale).

En caso de que el tamaño del área de aplicación, una vez transformada o no, resulte insuficiente para cubrir al objeto, la trama del bitmap se repetirá sobre éste tantas veces como sea necesario, hasta que toda la superficie quede "pintada". POV dispone del modificador map_type para permitirnos cambiar el tipo de aplicación del bitmap. Map_type va seguida de un valor con el que indicaremos este detalle; 0 es el valor por defecto y el que corresponde a una aplicación plana, 1 realiza un mapeado esférico sobre el objeto, 2 efectúa un mapeado cilíndrico y 5 un mapeado toroidal (para objetos con forma de donut). La forma del mapeado afectará a la apariencia del objeto.

Diseño elegido

Es fácil imaginar que las casas que ilustran estas páginas pueden modelarse de distintas formas. Se podría, por ejemplo, haber creado cajas de diferentes tamaños, y haber aplicado el bitmap a cada pared desplazando y rotando el plano de aplicación del mismo.

Este sistema, en concreto, fue considerado y finalmente desechado por parecer poco flexible y demasiado trabajoso. Había muchas partes cuya escritura habría de repetirse y paredes que existirían pero que no se verían, con la consiguiente pérdida de tiempo en cálculos inútiles. En lugar de esto, se optó por otro sistema bastante radical. Se decidió crear una colección de paredes o paneles diferentes que se emplearían para crear todas las casas. Un panel podía ser un muro con vigas cruzadas, una pared con puertas o ventanas, etc. De este modo, un montón de casas diferentes podía definirse con poco esfuerzo definiendo los modelos como uniones de paneles y otros objetos. Con esta filosofía una casa sencilla es, simplemente, 4 paredes a las que se les ha puesto un techo.

Un poblado medieval

En town.inc hay declaradas cerca de 50 estructuras utilizables de diversa complejidad.

Aunque se ha intentado seguir un cierto orden, observaréis que en las casas de distintos tamaños se emplean métodos diferentes. Esto se ha hecho así en parte por capricho, para averiguar que método era el más práctico, en parte por intentar ilustrar diferentes métodos para llegar a resultados similares y en parte porque un sistema que funciona bien en un tipo de objetos no tiene porque ser igualmente válido para otros.

La primera pared

Ante todo era preciso escoger un sistema de medidas para construir los objetos y calcular distancias. En Castlib, por definición, 10 unidades de POV equivalen a un metro. Otra norma elegida fue que el suelo, donde se colocarían los escenarios, estaría formado por el plano X-Z, a la altura Y=0 del origen de coordenadas.

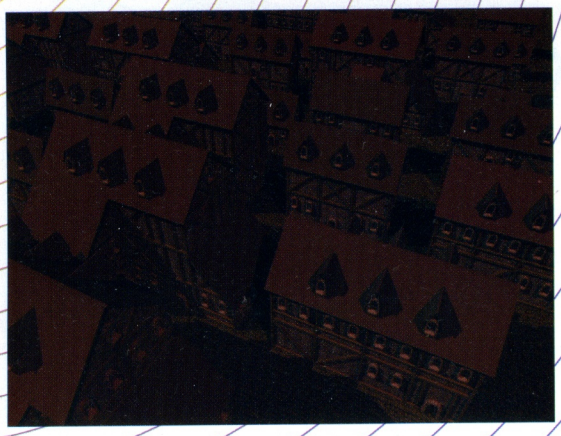
Definidas estas reglas, el primer objeto creado fue una sencilla caja que representaría un trozo de pared de 5x5 metros (50 pov-unidades por cada lado). A este primer objeto, al que luego se le añadieron vigas en los lados, se le llamó panelb (por panel básico). Como el bitmap se aplica por defecto en el plano X-Y, el panel fue colocado paralelamente a dicho plano.

Además, como se construirían nuevos paneles tomando el básico como base y como dichos nuevos paneles tendrían, a veces, que ser rotados para colocar las paredes laterales de las casas, panelb se definió centrado en el eje Z. Como las paredes son todas verticales (paralelas al plano X-Y) no era preciso centrar el panel inicial en Y, por lo que se colocó esta primera pared justo encima del suelo.

A esta pared inicial se le colocó el bitmap. Dicho bitmap, que no ha sido incluido con los fuentes de Castlib por no tener derechos de distribución sobre el mismo, es una imagen de una trama de rocas. Esta imagen es "tileable", como suelen decir los grafistas, o sea que las rocas de un lado encajan con las del extremo opuesto del bitmap. Esta característica es sumamente útil por dos razones: la primera porque, aunque se aprecie repetición, no se notarán fallos al aplicarse repetidamente la textura y la segunda porque no habrá que preocuparse de la posición en que se coloque ésta sobre el muro. La pongamos como la pongamos encajara.

De lo que sí había que preocuparse era del tamaño relativo que tendrían las rocas del bitmap sobre el muro. Como se dijo antes, el plano de aplicación de la textura se extiende por defecto sobre un área cuadrada de 1 unidad de largo. Esto significa que, como el muro tiene 50, la imagen se repetiría 50 veces por cada lado. O sea que no veríamos ladrillos sino tan sólo un feo punteado. Para remediar esto bastaba con

Aquí puede verse cómo un distinto vaor de seed sirve para generar calles diferente.



escalar el bitmap. Examinemos la declaración del objeto panelb:

```
#declare panelb=union{
  box{<-25,0,0>,<25,50,5>
pigment{  image_map{tga
"piedra2.tga" }
              translate <-.5,-
.5,0> scale <14,14,1> }}
      object{viga rotate
z*90 translate <0,50,0>}
      object{viga rotate
z*90 translate <0,2,0>}
      object{viga translate
<-25,25,0>}
      object{viga translate
<25,25,0>} }
```

Como vemos, panelb se ha declarado como una unión de objetos. El primer componente de la unión es la caja que hará las veces de muro y los 4 siguientes son las 4 vigas de madera que bordean el muro. (Las vigas son tan sólo cajas alargadas con texturas de madera). Después de la sentencia image_map siguen dos operaciones de transformación. La primera, translate, no tiene demasiada importancia y sirve tan sólo para que, al centrar el área de aplicación de la textura en <0,0,0>, la aplicación de la misma quede simétrica sobre el muro. La siguiente define el factor de escala de la aplicación en los ejes X-Y. Un valor mayor dara ladrillos más grandes. Si después de scale, por experimentar, hubiésemos aplicado la sentencia rotate zx45, el resultado habría sido la ladrillada diagonal que puede verse en una de las imagenes.

Más tipos de muros

Una vez definido el panel básico, podemos crear un nuevo panel basándonos en el ya creado. Por ejemplo:

```
// con puerta a la izq. y
ventanita tipo 1 a la der.
  #declare panel11=union{
      difference{
        object{panelb}
          object{recpuer1
translate <-14,0,0>}
          object{recven1
translate <12,25,-1>}
      }
      object{puerta transla-
te<-14,0,1>}
          object{marcove1
translate <12,25,2> }
  }
```

...se declara el panel número 11. Examinemos la declaración: Panel11 se define como una unión en la cual el primer elemento es una diferencia. El objeto de la resta es una copia de panelb que empleámos para crear la forma básica de panel11. O sea: las restas de los objetos recpuer1 y recven no se efectuan sobre el panelb original, sino sobre la copia del mismo que toma panel11, tan pronto como se halla la sentencia object{panelb}. Esta copia se encuentra en la misma posición y orientación espaciales y hereda las mismas propiedades (tamaño, materiales, etc.) que el original. Los objetos de resta empleados se hallan definidos al principio de town.inc y son agujeros para crear una puerta y una ventana. Estos agujeros son trasladados -ya que también estaban creados de forma centrada en el origen- a las posiciones donde irán los marcos de la puerta y la ventana. Los objetos-marco son los otros tres componentes de la unión.

(Nótese que los tres objetos situados dentro de las llaves de la operación difference NO son parte de la unión. El resultado de la operación de diferencia, en cambio, es un objeto del cual SI podemos decir que es un elemento de la union).

El nuevo objeto declarado; panel11, puede a su vez, ser empleado en la declaración de nuevos objetos (esta potente característica de POV nos ahorrará muchas líneas de texto). Podemos comprobar este detalle fijándonos en la declaración del panel12.

```
// con puerta a la izq.,
ventanita tipo 1 a la der. y
viga diagonal
  #declare panel12=union{
      object{panel11}
        object{viga scale
<1.5,1.33,1> rotate z*45
translate <0,25,-0.2>}
  }
```

Con tan sólo tres líneas declaramos el panel12. En dicha declaración se crea un nuevo objeto que es la unión entre una copia del panel11 y una viga cruzada diagonalmente sobre el panel.

Trucos

Una vez definidos los 15 paneles que forman nuestra colección, y con los que se han definido todas las casas de town.inc, es el momento de crear nuestra primera casa. Antes los pov-novatos deberían tomar nota de algunos trucos sencillos:

1) Casi siempre, el objeto debe construirse centrado en el origen de coordenadas.

2) Recordad que si un objeto está situado a cierta distancia del centro de coordenadas es muy fácil crear un nuevo objeto -idéntico- en una posición comprendida dentro del círculo que forma el radio desde la posición del objeto original hasta el origen de coordenadas. Este truco se empleó, por ejemplo para colocar las aspilleras de las torres circulares (ver PCmanía 47). Primero se definía una aspillera a x distancia del origen y luego las de-

más se definían haciendo una copia rotada x grados con respecto a la original.

3) Es muy útil colocar cámaras en los ejes que apunten al origen de coordenadas, donde hemos colocado al objeto en construcción, de este modo tendremos las vistas: frontal, lateral, superior, etc. Si la cámara apunta al centro del objeto será más fácil detectar imperfecciones en la simetría y otros detalles.

4) También es muy útil definir un plano con checker como suelo donde se colocarán los objetos que vayamos a construir. De este modo, podremos ver si las medidas y distancias entre objetos son correctas. En Castlib el metro equivale a 10 unidades por tanto deberemos escalar el checker por 10 para que cada cuadro equivalga a uno de nuestros metros virtuales.

Levantar casas

Se han empleado leves variaciones en la idea básica para crear los distintos tipos de casas. Sin embargo, el método ha sido siempre el mismo: definir uniones de objetos-muro, colocados y rotados convenientemente. Normalmente, cuando la casa iba a tener más de un piso, se procedía a declarar pisos y luego la casa se definia como una unión de pisos diversos. El segundo piso se colocaba (translate <0,50,0>) a 5 metros de altura (Y) sobre el primero, el siguiente a 10 metros, etc. Naturalmente, también se han empleado otros objetos; los marcos para ventanas y puertas, por ejemplo, se crearon empleando la sentencia Prism descrita en PCmanía 47. Las buhardillas son casas -que emplean panelb en todos sus muros- a las que se han aplicado operaciones de escalado y sobre las que se ha puesto una ventana sin escalar. Los tejados... Lo importante es que para usar town.inc, el usuario sólo tendrá que colocar la necesaria sentencia #include en su fichero, y citar desde allí los objetos cuyo nombre empieza por "casa". Las casas de 1 a 10 tienen 10 metros de altura, las de 20 a 29 tienen un piso extra, y así sucesivamente. Si el lector examina town.inc observará ocasionalmente identificadores con nombres tales como casa32a, casa34c, etc. Estos nombres indican que el objeto es una variación del diseño original.

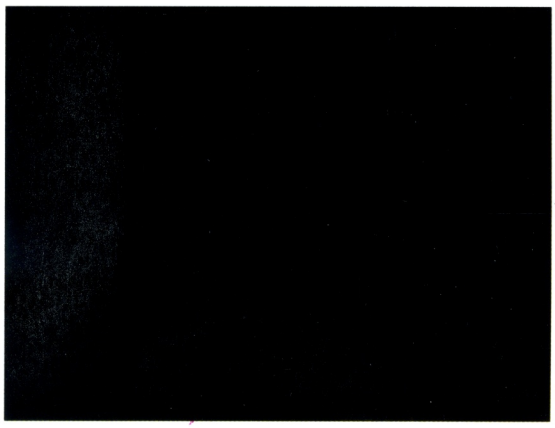
Levantar calles

Una vez listos para crear el pueblo surge el problema de qué método emplear. Podemos colocar las casas a mano pacientemente pero habrá que hacer muchas pruebas, ya que seguramente cometeremos muchos errores en la creación de las calles; casas que se superponen unas a otras, excesiva distancia entre las mismas, etc. La verdad es que el método más sencillo se basa en emplear algunas de las nuevas sentencias de POV 3.0. No vamos a describir ahora estas sentencias (ya lo comentamos en PCmanía 47), pero si diremos que aquellos que tengan nociones de programación no tendrán ningun problema con ellas. En el fichero generador de las povciudades se emplen dos bucles #while para crear una rejilla de casas. La colocación de una casa u otra se decide usando el generador de numeros pseudoaleatorios que incorpora POV (funciones Rand() y Seed()) y la función int(). También se emplean sentencias Switch y sus correspondientes #case. Todas estas sentencias tienen un uso casi idéntico al que podríamos esperar en un lenguaje de programación. Hay que subrayar un detalle muy atractivo; cambiando el valor dado como semilla a la función seed(), los valores suministrados por Rand() cambiaran y el resultado será un pueblo totalmente diferente.

Ciudades

Podríamos imaginar que las ciudades que podemos crear pueden tener cientos, o miles de edificios pero esto no es posible. Aunque los ficheros escénicos ocupan muy poca memoria, el modelo interno que construye POV para generar las escenas si necesita bastante RAM. Os aconsejamos que no creéis declaraciones de bucles while de objetos, ya que entonces se requerirá bastante más memoria al necesitarse espacio para el modelo declarado y la copia invocada. Desconocemos los detalles exactos acerca del modo en que POV gasta memoria. Tan sólo podemos repetir lo que hemos visto en las pruebas.

En cuanto a las fotos, no hay mucho que destacar. Básicamente sólo se usaron dos cámaras para todas ellas cambiándose casi siempre, en cambio, el valor de seed y el número de columnas y filas. Notad el fallo cometido en una de las fotos. La rutina extiende la ciudad desde un punto alejado de la posición <0,0> de X y Z. Como en dicha foto se utilizó una esfera para simular el mundo, y sólo tenía 100.000 unidades de radio, el resultado fue que las casas parecen flotar en el aire.



Edita: Hobby Press S.A.
Redacción:
José Manuel Muñoz
Edición y Diseño:
Equipo Pcmanía.
Depósito Legal
M-34844-92

Este suplemento se incluye conjunta e inseparablemente con PCmanía.

pcmanía

15 años
HOBBY PRESS